The relenc package

Lars Hellström*

October 2, 2010

Contents

1 Motivation **2**

2 Usage **6**
2.1 Author usage 6
2.2 Font designer usage 7
2.2.1 Some technical background 7
2.2.2 Defining variants of font-dependent commands 10
2.2.3 Defining variants of compositions 12
2.2.4 Defining compositions of variants 12
2.2.5 Setting the family search path 13
2.2.6 Where to put it all 13
2.3 Encoding designer usage 14
2.4 Power user commands 16
2.4.1 Debugging assistance 16
2.4.2 The 'define first' mechanism 17

3 Implementation **18**
3.1 Initial stuff 18
3.2 Variables and similar things 18
3.3 Search mechanisms for variable commands 19
3.3.1 Identifying the search path and initiating the search 19
3.3.2 Performing the search 21
3.3.3 Miscellaneous search-related macros 24
3.4 Making variable text commands 26
3.4.1 Miscellaneous support macros 26
3.4.2 Declaration commands 26
3.4.3 Definition commands 28
3.5 Making compositions of font-dependent commands 30
3.5.1 Commands for compositions that are variable 30
3.5.2 Commands for variants with compositions 33
3.6 Miscellanenous commands 35
3.6.1 Search paths 35
3.6.2 The define first mechanism 36
3.7 Debugging 36

*E-mail: Lars.Hellstrom@math.umu.se

3.8 Fix for a bug in LaTeX 37

The T1R encoding **38**

A Specification **38**
A.1 The coding scheme 39
A.2 The syntactic ligatures 39
A.3 The font dimensions 39
A.4 The font-dependent commands 39
A.5 On hyphenation patterns for the T1R encoding 43

B Implementation **46**

The zcm example font family **51**

C The zcm font family and reldemo.tex **51**

D Implementation **53**
D.1 Font definition file 53
D.2 The ecsubzcm package 54

1 Motivation

This paper is about some shortcomings that, in my humble opinion, exists in the way LaTeX handles fonts. I also point out a way in which these shortcomings can be overcome.

The primary problem is ligatures, but as there are a few different ligature concepts that are of interest, let me begin with specifying my terms. A *ligature* is a sequence of characters (almost always letters) that have been given an appearance somewhat different from the one the characters would have if simply put side to side, almost always because they would otherwise not look very pleasing to the eye. Despite this difference in appearence, it is still meant to be read as the entire character sequence, not as a completely new character. The canonical example of this is the 'fi' ligature.

In TeX fonts, there is a special mechanism to implement this, and everything that is implemented using this mechanism will be called *font ligatures*. It is almost always the case however, that some font ligartures are not ligatures as defined above, but simply a handy way to type characters that are hard or impossible to type using a standard keyboard; the canonical example of this is the '`--`' (two hyphens) to '–' (endash) conversion that is present in most TeX fonts. Such nonproper ligatures will be called *syntactic ligatures*, and proper ligatures will sometimes be called *aestetic ligatures* to stress their origin.

A *font-dependent command* in LaTeX is a command whose actions depend directly or indirectly on which font is the current. (I would not consider a command `\foo` defined by

```
\def\foo{\char65 }
```

as a font-dependent command since it always does the same thing. The results need not always be identical, but that is because the command is executed under different conditions.) An example of a font-dependent command is `\"`, which is (roughly) `\accent 127` when the current font is `OT1`-encoded and `\accent 4` when the current font is `T1`-encoded. (The dependence is indirect since the command directly depends on a macro which is set during the font selection process, but there is a dependence.)

For the purposes of this paper, if would also suffice to define a font-dependent command as a command that is defined by some of the commands `\DeclareTextCommand`, `\ProvideTextCommand`, `\DeclareTextSymbol`, `\Declare TextCommandDefault`, `\ProvideTextCommandDefault`, or `\DeclareTextAccent`. LaTeX documentation uses the term 'encoding-specific command' for these, but for reasons that will soon be appearent, that term would be somewhat inappropriate here.

Thus, with these definitions taken care of, it is now time to get to the point.

The recommended latin font encoding these days is the `T1`/'Cork'/'Extended TeX text' encoding, and this is rightfully so. It is clearly superior to the old `OT1` encoding, as it adds more than a hundred accented characters to those which can be used to form a word that TeX can automatically hyphenate, but there is at least one case in which the `OT1` encoding is preferable. This case is when the font has many ligatures.

In the `T1` encoding, there are seven slots available for ligatures, and these have been assigned to the 'ff', 'fi', 'fl', 'ffi', 'ffl', 'IJ', and 'ij' ligatures. Since all slots have been assigned to something, there is no place to put an additional ligature, even if it is needed. Thus the conclusion is that if a font is to be `T1` encoded, it cannot contain any ligatures in addition to the aforemensioned; to put it the other way, if a font design requires the presence of a ligature other than the aforemensioned, it cannot be `T1` encoded.

In the `OT1` encoding, there are only five slots assigned to ligatures, but there are 128 unassigned slots that can be used for anything the font designer wants. Thus having more than five ligatures in an `OT1` encoded font is no problem, but a recourse to using `OT1` is not a very good option, as it leaves the hyphenation problem unsolved. The solution, then, would seem to be the creation of a new encoding, and part of it will, but this will not be quite sufficient for reasons I will shortly describe.

For the moment though, let us, as an intellectual experiment, assume that we shall solve this problem with `T1` having too few slots for ligatures by creating a new encoding for a hypothetical font that would need more than seven ligatures. Let us also assume that the new encoding shall be a modified version of the `T1` encoding, where some accented characters will have been left out to make room for the ligatures. Finally, let us assume that we want to be as international as possible and include as many of the accented characters as we can squeeze in. These are three simple assumptions, and there are good reasons for all of them.

How *many* slots do we need to assign to ligatures, then? This varies, of course, between different font families, but it might vary *even more* between fonts in the same family. The `it` shapes might need a few more than the `n` shapes, while the `sc` shapes might not need any at all ('FI' (`fi`) and 'FI' (`f{}i`) look exactly the same in most font families). Instead, there are some accents which are harder to put on in the `sc` shapes (in many font families the ring on A in Å should touch the main letter; this is not what the default definition does), so it appears that

the optimal thing to do would be to have slightly different encodings for different fonts, even if they belong to the same family. This is theoretically no problem; TEX's macro facilities are flexible enough to allow user level commands that do different things in different fonts. It becomes, however, a problem to do this in a reasonably universal way, so that the macros produced work in general and not only for a single font family.

Standard LATEX has a mechanism for doing precisely this. Using the commands `\DeclareTextCommand`, `\DeclareTextSymbol`, `\DeclareTextAccent`, or one of their relatives, one can give a definition of a command that is used with one particular font encoding and not with any other. The problem with using this mechanism here is that one might have to have the normal and italic variants declared as having different encoding attributes (as well as different shapes), so one would have to either device a whole new set of font changing commands or redefine LATEX's own high-level font changing commands (such as `\textit`) to change encoding as well as shape or series. Neither alternative is good, and one can expect several incompability problems to arise for both of them.

A better solution starts with recognizing that there are actually two different 'encoding' concepts that can be found here. One is the attribute by which fonts are selected in LATEX, the other is the actual layout of a font. I will call this latter concept a *coding scheme* and reserve *encoding* for the former. (Formally, one may start by defining a *slot* to be an integer in the range 0–255 and a *glyph* to be a pattern (usually recognicable as a letter, digit, punctuation mark, or some other part of written language, but it need not always be). A coding scheme can then be defined as a mapping of slots to classes of glyphs. A font complies to a particular coding scheme if, for every slot n in the domain of the coding scheme, the glyph occupying slot n of the font is a member of the class that the encoding scheme maps n to. But I digress.) As far as I know, there is no strict defintion of what an encoding is, apart from the operational given in [3] as something that is part of the specification of a font. (The canonical source for such a definition would be [4], but that paper is, according to its author, "still in an embryo state".) In font discussions, an encoding is often taken to imply a specific coding scheme, and many encoding definition files seem to be all about listing the coding scheme, but is this implication suitable? I would claim that in this case, it is not.

A more constructive definition would be to see an encoding as a specification of which font-dependent commands are available to the author. An encoding definition file, on the other hand, is a specification of the interface between LATEX macros and the information in a TEX font. It does not matter to the author whether ő is `\char174` of the current font, generated as `\accent125o` by TEX, or whatever. The only thing that matters is that when the author types `Erd\H{o}s`, it comes out as Erdős.

Consequently, there is really no need for the font-dependent commands in LATEX to do the same thing for any two fonts with the same encoding attribute, it is merely the case that standard LATEX does not offer an interface for defining font-dependent commands in any other way. The natural remedy for this then, would be to write a package which offers such an interface. This is what I have done; the package is called relenc and this paper is its documentation. Its usage and implementation are described in the following sections, and the appendices describe some accompanying files.

I shall however conclude this section by an attempt to elaborate the above view on what an encoding is, or perhaps rather, what it should be.

The encoding property of a font is a set of rules that determines how the author's manuscript is interpreted—the input character `q` for example has not the same interpretation in a `T1` encoded font (where it is the letter 'q') as in an `OT2` encoded font (where it is a cyrillic letter whose closest latin equivalent is the Czech 'č'). An encoding specification should therefore be a formalization of an agreement between the font designer on one hand and the author on the other—it specifies which rules each side must comply with and which results that can then be expected. An example of the author's rules may be to refrain from writing TEX code like `\char 166`, because the font designer may have an option on what to put in that slot. If the author breaks the rules, he or she may find that the manuscript produced contains text whose meaning is not the same if typeset with two different fonts even if they do have the same encoding property. In practice, the author's rules for the standard text encodings are pretty much the same as the rules on how write TEX code we find in every elementary book on the subject, so they are hardly new to us.

An example of the font designer's rules may be to put an exclamation mark in slot 33, so that `!` actually print as one, or to include a font ligature that converts two consequtive hyphens to an endash, so that `--` actually will print as an endash, which the author by tradition expects it to do. If the font designer breaks the rules then authors who follow their rules might find that they do not get the right results anyway and such a font designer is likely to get complaints from authors about this. In practice however, the font designer rules are often vaugely specified if specified at all and hence there are gray areas for most encodings where there are no rights and wrongs. The `OT1` encoding is probably the one most plauged by these; the dollar versus sterling problem (an excellent example of how changing the glyph of a single slot many completely alter the interpretation of a text) is a classic. One of my intentions with writing this text is to work for that these gray areas are shrunken or even completely eliminated, although I do not think there is anything that can be done for the `OT1` encoding—its irregularities are much too well known and exploited.

Now if an encoding is (a formalization of) an agreement, how do the parties agree to it? On the font designer's side this happens when the font designer gives a font a specific encoding by writing a font definition file that defines that font with that encoding. On the author's side this happens when the author selects a font with that encoding property.

So far the informal description, now it is time to get to the formalization. Which exactly are the rules for the author and for the font designer? This varies between different encodings, but only in the details. The areas the encoding specification must cover can be listed and are:

- Which input characters that can be used directly to produce some of the font's glyphs in the output and what they will generate. This pertains to the author, who shouldn't use other input characters. The allowed ones do however have well-defined results.

- Which coding scheme the font must comply with. The pertains to the font designer. There are no direct restrictions on the use of slots not listed in this coding scheme.[1]

[1] There may be indirect restrictions, see below.

- Which the required syntactic ligatures are. This pertains to both author and font designer. The author cannot trust any in addition to these, the font designer must include them.[2]
- Which the font-dependent commands are and what they will generate. This pertains to the author in the same manner as does the input character rules.
- Which the required font dimensions are and what they stand for. This pertains to both the author and the font designer in the same manner as does the syntactic ligature rules.[3]

After these have been specified, the grey areas should be very small indeed! There are however a few additional twists that must be sorted out.

If the required coding scheme listed in the encoding specification does not cover all the 256 slots, then one must be aware that in particular the required syntactic ligatures, but also the font-dependent commands, may impose some restrictions on the font's coding scheme in addition to those expressed by the given coding scheme that the font must comply with. These restrictions are then of the form that a glyph from a specific class must be assigned to some slot, but the font designer may freely choose exactly which slot. Thus any single slot not specified by the required coding scheme may be used for just about anything.

The use of the relenc package requires that the following area has to be added the ones listed above.

- The font designer must see to that for every combination of a variable command and a font, there is a variant that will give the specified result.[4]

Hyphentation patterns do also offer theoretical problems to the use of the relenc package, as these refer explicitly to the coding scheme of the font. Problems with these can however not result in anything worse than bad hyphenation, so the interpretation of a text should not be affected. It is furthermore the case that in practice the problems can often be avoided (see Subsection A.5).

Finally, there are two font parameters—`\hyphenchar` and `\skewchar`—that do explicitly relate to the coding scheme of the font and which are not stored in the font itself. It is possible that the value of at least one of these should be specified in an encoding specification, but that particular question is not of immediate interest to the relenc package, as LaTeX itself already provides the font designer with the ability to set these for each font individually (using the sixth argument of `\DeclareFontShape`).

2 Usage

2.1 Author usage

All the author has to do to use fonts with a relaxed encoding, as opposed to fonts with for example the `T1` encoding, is to include the command

```
\usepackage{relenc}
```

[2] It could well be that there *should not* be any syntactic ligatures in addition to these. I know of no situation where there would be an advantage in adding syntactic ligatures.

[3] Even though very few physical authors access any font dimensions, the same does not hold for packages, and these also count as authors in this context.

[4] The terms *variable command* and *variant* are explained in Subsubsection 2.2.1.

in the preamble and load the encoding definition file, for example using the fontenc package. It is however important that the relenc package is loaded *before* the encoding definition file, as the latter uses commands defined in the former.

2.2 Font designer usage

For a font designer, it is important to know at least in broad outline how the mechanisms made available through the relenc package work, which is why this subsection starts with a description of that. There is however a convention followed in the remainder of this paper that the reader should be aware of and this convention has to do with how control sequences are written.

In this paper, there are many control sequences with "strange" names, meaning names that mixes letters and non-letters in pretty arbitrary ways, so that these names cannot be read as one normally reads TEX code. Therefore thin spaces are inserted around names of control sequences, regardless of whether a space character at that place would automatically be skipped by TEX while it is reading the code or not. A space character that is really meant to "be there" will be written as a visible space (␣). All control sequences will, as usual, be written with an opening backslash, but this backslash is not part of the name of the control sequence.

Excepted from the above convention about spaces is the actual TEX source code for that appear in Section 3 and onwards (the lines of this is numbered, so it should be easily distinguishable) and some pieces of "alternative" source code in the same sections. These exceptions should be easy to recognise for the readers who are interested in that particular material.

2.2.1 Some technical background

The main feature added by the relenc package is that of the *variable commands*; it is through making commands variable that their definition may depend on which font is the current. This is not how TEX would see it, since the definition of a variable command (as a TEX control sequence) actually does not change after the command has been made variable! Rather, a variable command is a macro which expands to different things depending on which the current font is.

With overwhelming probability, this is something you have encountered before, although you might not have realised it. Under LaTeX 2_ε, all accenting commands and all commands for letters other than a–z (such as æ, ø, and ß) are like this. The only difference lies is what will affect the eventual outcome of the command. The LaTeX 2_ε kernel only supports dependence on which the current encoding is. The variable command concept makes dependence on the current family, series, and shape possible as well.

Both systems are quite similar in that they rely on `\csname` lookups. What happens to, for example, the command `\foo` is the following: First it gets `\string`ed. This converts the single control sequence token to the sequence of character tokens which would form the name of the command; in this case to `\`, `f`, `o`, and `o`. Then a piece of text is put in front of that character sequence, and finally the result of that is taken to be the name of a new control sequence. This process mainly generates control sequnces with very peculiar names; if the bit of text is, say, `T1` then the new control sequnce will be `\T1\foo` (this is *one* control sequence). Such names are impossible to type without a lot of trickery, but that

is deliberate, since they should not be accessed directly. If the control sequence thus formed is defined, then the definition of that control sequence will be taken as the intended definition of the control sequence `\foo` it all started with.

The systems differ in what pieces of text they put before the name of the command and what they do if the control sequence formed is not defined. The LaTeX 2_ε kernel starts by using the name of the current encoding as the prefix text. If that fails, it tries with `?` instead. If that fails too, an error message is issued. The variable commands defined using the relenc package have a more general approach.

This approach relies on the concept of a *search path*, about the structure of which more will be said later. For the moment, it is sufficient to say that it consists of a sequence of *blocks* of text. The looking up process consists of a loop in which the following is done: The first block is removed from the search path, and the text it contains is used to form the name of a control sequence, as described above. The rest of the search path is saved away as the *remaining search path*. If the control sequence formed is defined, then it is used as the definition of the command, and if it is not, then the process is repeated. When the process is repeated however, it starts by removing the first block of the remaining search path, which is the second (or third, or fourth, depending on which iteration of the loop is the current) block of the entire search path, while once again saving the rest as the new remaining search path. Not until the entire search path has been scanned in this way will an error message be issued.

This means arbitrarily many possibilities can be tested in searching for the definition of a command, but about six is probably a realistic upper bound on how many there will be in practical applications. In many cases it will be even fewer.

What has been mensioned so far does not mean there necessarily is any dependence on the current font, but it opens the possibility. The trick is that the pieces of text, which are the blocks in the search path, can contain not only characters but also macros (and other expandable stuff)—as long as everything eventually expands to character tokens, everything is fine. The point here is that the control sequences that contains the names of the current encoding, family, series, and shape—`\f@encoding`[5], `\f@family`, `\f@series`, and `\f@shape` respectively—are of this kind. Thus making the definition of a variable command depend on these attributes of the current font is simply a matter of making the corresponding control sequences part of the texts in the search path.

The above might give the impression that the variable commands are ment to be used instead of the encoding-specific commands of standard LaTeX, but that is not the case. What actually happens is that the control sequences of type `\T1\foo` that the LaTeX 2_ε kernel looks up will themselves be variable commands. This means that to LaTeX, the commands in a relaxed encoding whose definitions depend on the current font are just normal encoding-specific commands, even though they do a lot of peculiar things before they actually generate any typeset material, but on the other hand LaTeX doesn't care what they do, as long as it finds a definition.[6]

This has probably been a bit abstract, so an explicit example might be in place.

[5] For technical reasons, it is probably better to use `\cf@encoding` instead. See page 20 for a discussion of this.

[6] It also saves me a lot of work, since I won't have to bother with trying to make the variable commands robust—LaTeX already makes the encoding-specific commands robust.

Let's say that the current encoding is `T1R` (this is an existing relaxed encoding), the current family is `zcm` (this is an example family[7]), the current series is `m`, and the current shape is `n`. Furthermore let's say the user has just issued the font-dependent command `\foo` (this is not really a font-dependent command, but let's assume it is). What will happen?

1. The actual control sequence `\foo` causes the LaTeX 2_ε kernel to start look for a definition. It first tries `\T1R\foo`, then `\?\foo`, and if neither is defined then an error message is given. The case of interest here is that `\T1R\foo` is defined, because then the LaTeX 2_ε kernel is content and TeX will act as if `\T1R\foo` was issued instead.

2. If the final definition of `\foo` is to depend on family/series/shape then `\T1R\foo` must be a variable command. The first thing which happens then is that relenc starts looking for a search path to use. Search paths are stored in macros, and the names of these macros are formed in a manner similar to that in which the other lookup names here are formed.

   The two macros which can contain the search path are `\T1R/zcm-path` and `\T1R-path` (these are still only single control sequences), and they are tried in that order. If none of them exists, then an error message is given. The second of the two is common to all fonts using the `T1R` encoding and must be defined by the encoding designer. A font designer can choose to define a search path of his or hers own, and that will then be named as the first of the two above. A family specific search path completely overrides the encoding specific (the latter is in that case not even considered), but in many cases the encoding specific will do just fine.

   Let's assume that `\T1R/zcm-path` is defined and consists of

   {⟨*enc*⟩/⟨*family*⟩/⟨*series*⟩/⟨*shape*⟩}
   {⟨*enc*⟩/⟨*family*⟩/?/⟨*shape*⟩}
   {⟨*enc*⟩/⟨*family*⟩/?/?}
   {⟨*enc*⟩/?/?/?}

   (Each block is written on a separate line. The text of the block is everything between (but not including) the braces, which act as delimiters of the block. ⟨*enc*⟩, ⟨*family*⟩, ⟨*series*⟩, and ⟨*shape*⟩ denote the LaTeX macros listed above which contain the names of the current encoding, family, series, and shape respectively.)

3. Once the search path is found, it is scanned. In this particular case this means that the control sequences `\T1R/zcm/m/n\foo`, `\T1R/zcm/?/n\foo`, `\T1R/zcm/?/?\foo`, and `\T1R/?/?/?\foo` are tried in that order. If none of them is defined, an error message is given, but let's assume that `\T1R/zcm/?/?\foo` is defined and neither `\T1R/zcm/m/n\foo` nor `\T1R/zcm/?/n\foo` are. This corresponds to the case that there is a definition of the variable command that is specific for the family, but not any specific for the shape or series.

4. The final stage is that `\T1R/zcm/?/?\foo` gets executed.

[7] The `zcm` font family is described in Appendix C.

There are now only a few more things to sort out before the description of the commands a font designer has available can commence. Firstly, control sequences like the above `\T1R/zcm/?/?\foo`, that hold an actual definition of a variable command, are called *variants* of that command. The processing during the scan of the search path that is connected to one block in the search path is called a *step* in that scan.

Secondly, there is another thing which might affect the definition of a command, viz. the first argument of the command. Commands for which the first argument is checked before the actual definition is determined are called *composite commands*, or are said to be *composed*. The alternative definitions of them are called *compositions*. Each composition is used for exactly one value of the argument, and the main composite command contains a definition which is used for all values for which there is no composition.

This too is a mechanism that is present in the LaTeX 2_ε kernel, what relenc does is that it introduces some commands to make variable commands composed or vice versa. Very much like variable commands, composite commands rely on `\csname` lookups, but instead of adding a prefix to the command name, the composition mechanism adds a suffix consisting of a hyphen (`-`) and the first token of the first argument (as a precaution, this token is `\string`ed beforehand, to convert it to character tokens if it was not already).

An example of this, from the `T1` encoding, is the acute accent command `\'`. The command that actually is composed is `\T1\'`, which holds the definition of `\'` in the `T1` encoding, and one of its compositions are `\\T1\'-a`. This is a macro which expands to the letter á, which is the expected result of `\'{a}`. There is no composition for the argument `\ae` in the `T1` encoding, so if the user issues `\'{\ae}` the lookup mechanism finds nothing and the default definition is used, yielding 'æ'. Had there been a composition however, it would have called `\\T1\'-\ae`. Cases like these are why the `\string`ing precaution is necessary; most commands generate errors when TeX meets with them inside a `\csname` ... `\endcsname` pair.

With that description completed, it is now time to describe the usage and purpose of the commands available to the font designer. It should perhaps be pointed out that most of them are about defining variants of commands, as making a command variable lies within the powers of the encoding designer.

2.2.2 Defining variants of font-dependent commands

Among the arguments of every variant defining command is the sequence `{`⟨*encoding*⟩`}{`⟨*family*⟩`}{`⟨*series*⟩`}{`⟨*shape*⟩`}`, which specifies which variant of a command is being defined. The arguments should consist of letters and/or figures, but any combination of these parameter fields might be left empty. A field left empty signifies that the intended variant may be used regardless of what value that attribute may take. Thus `{T1R}{zcm}{m}{n}` is used when defining a variant specific to this encoding, family, series, and shape, whilst `{T1R}{zcm}{}{}` is used when defining a variant that applies for every font in the `T1R`-encoded `zcm` family. Technically, a field left empty will be filled with a question mark. Thus the `{T1R}{zcm}{}{}` variant of `\foo` will be stored in the control sequence `\T1R/zcm/?/?\foo`.

`\DefineTextSymbolVariant`
`\DefineTextAccentVariant`

`\DefineTextSymbolVariant` and `\DefineTextAccentVariant` are the two simplest commands for defining a variant. The former makes the variant output a single character, whose slot in the font is given as the argument ⟨*slot*⟩. The latter

should be used for variants of accent commands, as an accenting command is precisely what it defines. The character used for the accent is the one with slot number ⟨*slot*⟩. `\DefineTextSymbolVariant` and `\DefineTextAccentVariant` parallell the commands `\DeclareTextSymbol` and `\DeclareTextAccent` respectively that are found in standard LaTeX.

`\DefineTextCommandVariant`

If the above are not sufficient for the definition of a variant of some command (they are not, for example, general enough to define any of the accents put *under* letters), complete generality is offered through the `\DefineTextCommandVariant` command, which can be used to define any TeX macro. (It consists simply of a `\gdef` to the control sequence that stores the variant in question.) This means the ⟨*parameter text*⟩ should be formated as for the `\def` command, without any surrounding braces or such. Also notice that every token in the ⟨*parameter text*⟩ counts, including spaces and end of lines.

Apart from the arguments mensioned, all the above commands have an argument ⟨*cmd*⟩. This is the name of the base font-dependent command of which you want to define a variant. It is not the name of the actual variable command, so you should write `\foo`, not `\T1R\foo`. The syntaxes of the commands are as follows:

```
\DefineTextSymbolVariant {⟨cmd⟩} {⟨encoding⟩}
  {⟨family⟩} {⟨series⟩} {⟨shape⟩} {⟨slot⟩}
\DefineTextAccentVariant {⟨cmd⟩} {⟨encoding⟩}
  {⟨family⟩} {⟨series⟩} {⟨shape⟩} {⟨slot⟩}
\DefineTextCommandVariant {⟨cmd⟩} {⟨encoding⟩}
  {⟨family⟩} {⟨series⟩} {⟨shape⟩}
  ⟨parameter text⟩ { ⟨replacement text⟩ }
```

`\NewTextCommandVariant`
`\RenewTextCommandVariant`
`\ProvideTextCommandVariant`

relenc does also offer some `\newcommand`-style commands for defining variants of font-dependent commands, for font designers who prefer that. They do offer some additional functionality, as they can make commands which take an optional argument, but I am not currently aware of any font-dependent command that uses this feature. One reason the feature is offered is that variable command processing comes before optional argument processing, hence if a variable font-dependent command can have an optional argument then all its variants must be able to cope with that argument when it is present.

Technically the commands boil down to an application of `\newcommand`, `\renewcommand`, or `\providecommand` respectively (the starred forms, to be exact). Thus you may get error messages if the variant is already defined or not defined, depending on which command you use. As the error messages are the standard LaTeX error messages, they may be somewhat confusing. Still, a somewhat confusing error message may be better than none at all.

```
\NewTextCommandVariant {⟨cmd⟩} {⟨encoding⟩} {⟨family⟩}
  {⟨series⟩} {⟨shape⟩} [⟨numargs⟩] [⟨default⟩]
  {⟨replacement text⟩}
\RenewTextCommandVariant {⟨cmd⟩} {⟨encoding⟩}
  {⟨family⟩} {⟨series⟩} {⟨shape⟩} [⟨numargs⟩] [⟨default⟩]
  {⟨replacement text⟩}
\ProvideTextCommandVariant {⟨cmd⟩} {⟨encoding⟩}
  {⟨family⟩} {⟨series⟩} {⟨shape⟩} [⟨numargs⟩] [⟨default⟩]
  {⟨replacement text⟩}
```

2.2.3 Defining variants of compositions

As compositions can be variable, there are commands for defining variants of them. The situation here is simpler than for font-dependent commands in general since compositions cannot have any arguments, consequently there is no need to provide such a large variety of definition commands as for defining command variants.

\DefineTextCompositionVariant
\DefineTextCompositionVariantCommand

The most important is `\DefineTextCompositionVariant` which corresponds to `\DefineTextSymbolVariant`—it makes a variant which simply typesets one of the characters in the font. The most general command is `\DefineTextCompositionVariantCommand` which defines the variant to be a parameterless macro without other restictions.

\DefineTextUncomposedVariant

A special, but probably rather common macro to define a variant of a composition to be, is the macro consisting of the noncomposite definition applied on the argument for the composition, because defining the variant this way is probably the easiest way to free a slot in the font for other purposes. Hence there is a special command for doing this: `\DefineTextUncomposedVariant`. It resembles the other two, but there is of course no argument that gives the definition of the variant and there is a special restriction, namely that the ⟨*encoding*⟩ argument must not be empty!

The arguments of these commands are as for the commands for defining variants of font-dependent commands, except for one designated `{`⟨*argument*⟩`}`. This is the argument which is passed to the font-dependent command that corresponds to the current composition—the composition of which a variant is to be defined.

```
\DefineTextCompositionVariant {⟨cmd⟩} {⟨encoding⟩}
  {⟨family⟩} {⟨series⟩} {⟨shape⟩} {⟨argument⟩} {⟨slot⟩}
\DefineTextCompositionVariantCommand {⟨cmd⟩}
  {⟨encoding⟩} {⟨family⟩} {⟨series⟩} {⟨shape⟩} {⟨argument⟩}
  {⟨replacement text⟩}
\DefineTextUncomposedVariant {⟨cmd⟩} {⟨encoding⟩}
  {⟨family⟩} {⟨series⟩} {⟨shape⟩} {⟨argument⟩}
```

2.2.4 Defining compositions of variants

Things can be done the other way round too—a variant of a font-dependent command may have compositions. These compositions are then completely independent of any compositions of the base font-dependent command. Unlike compositions of a font-dependent command (which must be *declared* in the encoding definition file and are common to all fonts in a particular encoding), compositions of a variant can be *defined* whenever a variant can be defined. Thus making compositions of variants lies within the powers of the font designer.

\DefineTextVariantComposition
\DefineTextVariantCompositionCommand

There are two commands for defining compositions of variants: `\DefineTextVariantComposition` and `\DefineTextVariantCompositionCommand`. The difference between them is simply that the latter command defines the composition to be a macro with the given replacement text, while the former defines it to be a chardef token for the given slot. What is more interesting is what these commands do if the variant they are to make a composition of is not defined, because in this case they define the default definition to be a macro that resumes the scan of the search path. This means that the font designer can choose to specify some compositions early in the search path and others later—and perhaps more

importantly—can give special definitions for some compositions early in the search path without having to copy the default definition to that level.

As it happens, the names of the control sequences, in which the definitions of compositions of variants and variants of compositions respectively are stored, are slightly different (a backslash appears at different positions). Hence it is possible to have both for exactly the same ⟨*encoding*⟩ ⟨*family*⟩ ⟨*series*⟩ ⟨*shape*⟩ ⟨*argument*⟩ combination for a composition of variant and variant of composition without having them overwriting each other, although there is hardly any point in having things set up this way.

```
\DefineTextVariantComposition {⟨cmd⟩} {⟨encoding⟩}
  {⟨family⟩} {⟨series⟩} {⟨shape⟩} {⟨argument⟩} {⟨slot⟩}
\DefineTextVariantCompositionCommand {⟨cmd⟩}
  {⟨encoding⟩} {⟨family⟩} {⟨series⟩} {⟨shape⟩} {⟨argument⟩}
  {⟨replacement text⟩}
```

2.2.5 Setting the family search path

Setting the family search path is pretty straightforward: The search path is the last argument, encoding and family in question the two other. A useful feature here is that inside the search path argument, `@` will be a letter and all spaces and newlines are ignored. This means the example search path from Subsubsection 2.2.1 can be set even by a command call as spaced out as the following

```
\SetFamilySearchPath{T1R}{zcm}{
   { \cf@encoding / \f@family / \f@series / \f@shape }
   { \cf@encoding / \f@family / ? / \f@shape }
   { \cf@encoding / \f@family / ? / ? }
   { \cf@encoding / ? / ? / ? }
}
```

and even if it appears in the preamble of a document (this is handy when debugging a font family).

Search paths *must* be set using the `\SetFamilySearchPath` or `\SetEncoding SearchPath` commands, otherwise the case that no definition of a variable command is found cannot be handled correctly, with the effect that TEX gets hung in an infinite loop.

```
\SetFamilySearchPath {⟨encoding⟩} {⟨family⟩}
  {⟨search path⟩}
```

2.2.6 Where to put it all

One topic that has not been delt with above is where the font designer is to put all these commands for defining variants and setting search path. In my opinion, there is only one possible place—the font definition file[8]. This is also the logical place to put the commands, since this is the file in which the font designer describes

[8] I am well aware of the rules for which commands may be used in font definition files that are described in [3]. I have however chosen to disregard from these rules in the case of commands defined by the relenc package, as this case could hardly have been foreseen by the prescribers of these rules.

his or her font family to LaTeX. In particular, one cannot expect full functionality if the commands are put in a package, since it is perfectly possible to select a font without using a standard package for this.

Of course, definition commands can also appear in an encoding definition file and anything that can appear in an encoding definition file may also appear in a package file, even though packages containing such code are often of a rather special nature.

2.3 Encoding designer usage

The encoding designer's work in making a relaxed encoding is very much like the work in making a normal encoding. There are only two additional steps: It must be decided which commands and compositions that should be variable, and an encoding search path must be set. Both of these are more a matter of planning than writing TeX code, but it seems best to treat the coding first.

Each of the commands for declaring a variable font-dependent command or composition corresponds to a command for declaring a non-variable font-dependent command which is part of standard LaTeX, as is shown in the following table. The correspondence is not one to one, but it is pretty close.

Standard declaration command	Variable declaration command
`\DeclareTextCommand`	`\DeclareTextVariableCommand`
`\DeclareTextCommand`	`\DeclareTextVariableCommandNoDefault`
`\ProvideTextCommand`	`\ProvideTextVariableCommand`
`\DeclareTextSymbol`	`\DeclareTextVariableSymbol`
`\DeclareTextAccent`	`\DeclareTextVariableAccent`
`\DeclareTextComposite`	`\DeclareVariableTextComposition`
`\DeclareTextCompositeCommand`	`\DeclareVariableTextComposition`

`\DeclareTextVariableSymbol`
`\DeclareTextVariableCommand`
`\ProvideTextVariableCommand`
`\DeclareTextVariableAccent`
`\DeclareTextVariableCommandNoDefault`

The difference between on one hand the commands `\DeclareTextVariableSymbol`, `\DeclareTextVariableCommand`, `\ProvideTextVariableCommand`, and `\DeclareTextVariableAccent` and their non-variable counterparts on the other is that the font-dependent command they declare will become a variable command, while the definitions given will be used to define the encoding-level variant of the command. The arguments are exactly the same as for the commands' non-variable counterparts of standard LaTeX.

The `\DeclareTextVariableCommandNoDefault` command only declares a font-dependent command and makes it variable, but does not define any of its variants. This can actually be useful if one is writing an encoding that is a relaxed version of another encoding, such as the `T1R` encoding, since many commands will have the same encoding-level definition in both encodings. It is then possible to include the name of that other encoding in the search path, so that the same control sequence will hold the definition of a command in two different encodings.

`\DeclareVariableTextComposition`

The `\DeclareVariableTextComposition` command declares a composition of a command, like `\DeclareTextComposite` or `\DeclareTextCompositeCommand`, and makes that composition variable. But it is also like `\DeclareVariableTextCommandNoDefault` in that it does not define any variant of the composition. To define a variant, one of the commands in Subsection 2.2.3 must be used as well[9].

[9] I am not sure that this is a good way to organise it. Perhaps there should be commands combining these functions.

`\DeclareVariableTextComposition` takes three arguments: the command, the encoding, and the argument for which a composition is to be declared.

```
\DeclareTextVariableSymbol {⟨cmd⟩} {⟨encoding⟩}
  {⟨slot⟩}
\DeclareTextVariableCommand {⟨cmd⟩} {⟨encoding⟩}
  [⟨arguments⟩] [⟨default⟩] {⟨replacement text⟩}
\DeclareTextVariableCommandNoDefault {⟨cmd⟩}
  {⟨encoding⟩}
\ProvideTextVariableCommand {⟨cmd⟩} {⟨encoding⟩}
  [⟨arguments⟩] [⟨default⟩] {⟨replacement text⟩}
\DeclareTextVariableAccent {⟨cmd⟩} {⟨encoding⟩}
  {⟨slot⟩}
\DeclareVariableTextComposition {⟨cmd⟩} {⟨encoding⟩}
  {⟨argument⟩}
```

`\SetEncodingSearchPath`

This is very much like `\SetFamilySearchPath`; the main difference to setting a family search path is that a relaxed encoding *must* set its encoding search path.

```
\SetEncodingSearchPath {⟨encoding⟩} {⟨search path⟩}
```

Now to the part which is not coding. As noone, at the time this is written, is particularly experienced in the creation of relaxed encodings, this is not a guide of how to do that. This is only a collection of some observations I made when I created the `T1R` encoding and the relenc package.

- When making a relaxed encoding: If you mainly want to free some slots, so that you can include some new set of glyphs (for example additional ligatures) in the font, the obvious place to start is to reduce the number of slots that are assigned to compositions, by implementing these in a variable way.

- When relaxing a composition, there are two ways of doing this: making the composition variable, or making the command variable and defining a composition of some variant. The cost (i.e., the number of special definitions you have to make) is connected to different things in these methods.

  In the case of a composition of a variant, there is a cost connected to having a composition. In the case of a variable composition, the cost is rather connected to not using the default definition for the composition. In the usual case that one either uses a special glyph for a composition or uses the default definition of the accenting command, this means that composition of variant is cheaper if a minority of the compositions uses the default definition and variable composition is cheaper if a majority uses the default definition.

- In some cases, the default definition of an accenting command tends to be suitable for some font families, but inappropriate for others. An example from the `OT1` encoding is that the definition of `\c` starts by looking at the *height* (!!!) of the character it is to put a cedilla under. If the height is exactly 1 ex then the `\accent` primitive is used, otherwise the accent is put in place using a `\vtop` construction. This works fine (I suppose, trusting

DEK to have known what he was doing) for fonts with idealized heights and depths of characters, such as the Computer Modern family of fonts, but is a pure waste of time if the heights and depths are computed from the bounding boxes of the glyphs (like *fontinst* [2] does).

The conclusion of all this is that it might be a good idea to make accenting commands that have such a specific default definition variable, regardless of how any compositions of these commands might be implemented, so that font designers can override the definitions in case they want to.

Apart from this, there is not much advice I can give. It is however likely to be a good idea to try to make a specification of the encoding—like described in Section 1 or in some other way, detailed or only in loose sketches—before starting to do the coding.

2.4 Power user commands

This subsection treats some commands that may be useful to advanced users of the relenc package (this includes all font and encoding designers); in any case, the novice author users can do perfectly well without using the commands described here.

2.4.1 Debugging assistance

`\ShowVariantSearchResult`

As the way from user level command to definition given is quite long if the command is variable, there are many instances in which things can go wrong. `\ShowVariantSearchResult` may help in sorting out what exactly happened. Its primary function is to print the contents of all internal variables in relenc on the terminal and then wait for a command, just like after the primitive TEX command `\show`. As an extra service, `\ShowVariantSearchResult` also prints the current encoding, family, series, and shape.

As most of the processing in relenc is done in TEX's mouth, there is not very much left to show. The most important piece of data there is is the *remaining search path*. This is the part of the search path that was *not* scanned in looking for a definition; by comparing it to the whole of the search path used, one can determine at which stage a definition was found. The other thing shown is the definition of `\RE@first@search@item`, which normally is defined to be a parameterless macro that expands to the first block in the search path most recently used. There are however two cases when it is not: (i) If a definition was found in the first stage of the most recent search, `\RE@first@search@item` is not altered. (ii) If the search has been restarted (see Subsubsection 2.2.4) then `\RE@first@search@item` is a macro with a parameter.

Despite these reservations, `\ShowVariantSearchResult` provides about all the information there is to get about what a search has found. It might also be instructive if you want to understand the inner workings of the relenc package in more detail.

`\ShowVariantSearchResult`

Should `\ShowVariantSearchResult` not give you enough information, you can of course always set `\tracingmacros` to 1 and `\tracingcommands` to 2 for the time it takes to execute the command you are trying to debug, this will give you the

	Default encoding		
	OT1	T1	T1R
Time taken using T1	253.4 s	197.5 s	255.2 s
Time taken using T1R (DFM on, no FSP)	339.7 s	349.5 s	294.2 s
Time taken using T1R (DFM off, no FSP)	446.4 s	458.3 s	400.5 s
Time taken using T1R (DFM off, has FSP)	334.4 s	350.9 s	293.2 s
Time taken using T1R (DFM on, has FSP)	316.7 s	327.5 s	269.9 s

DFM = Define First Mechanism

FSP = Family Search Path. The family search path used was optimised to examine only the levels at which there actually existed some variant.

The 'default encoding' in this table is the encoding whose encoding definition file was read in last. As explained in [1], this means that all commands declared in that encoding will execute somewhat faster when that encoding is the current.

The test text used consisted of all non-accented letters declared in the T1 encoding (a–z, as well as æ, ß, ı, and a few others) in both upper and lower case, as they are as well as accented with every accent command available (\‘, \’, \^, \~, \", \H, \r, \v, \u, \=, \., \b, \c, \d, and \k). These 32 lines were then repeated 100 times, to reduce the relaive amount of time taken to start the process.

Table 1: A comparision of typesetting speed

whole picture of what relenc does. Before attempting this drastic action however, you should familiarise yourself with the implementation of the relenc package.

2.4.2 The 'define first' mechanism

The 'define first' mechanism, which has not been mensioned until now because it is not really related to anything else in the package, is something very few users should ever have to bother with. It can however speed up the typesetting process, as demonstrated in Table 1.

What the define first mechanism (DFM) does, when it is active, is that if a definition is not found in the first step of the search path scan and a definition is found in some later step, then that definition is copied to the control sequence scanned in the first step. Thus the next time that the same command is issued, the scan of the search path will find a definition in the first step.

This can speed up the search considerably, but there is a price to pay: More control sequnces gets defined, meaning more of TEX's memory is being used for storing definitions of variable commands.[10] If you run out of memory while typesetting a document with the DFM on, turning it off will lower the memory requirements. If your TEX is generally low on memory however, you should probably not be using relaxed encodings at all, since the basic deal of the entire package is to

[10] Or so it would appear ... Some things I've recently learnt about how TEX's internal tables work has made me wonder about whether it really takes more memory, so I am currently not sure. Perhaps someone competent in the area of TEX's memory management will volunteer to sort things out for me?

loosen the restrictions on fonts for a particular encoding by increasing the number of control sequences needed for the typesetting process.

But these differences should be seen for what they really are, differences in speed for one of the many things TEX have to do to typeset something. TEX does no linebreaking during the tests in Table 1 (hence no hyphentaing either), does not read any input after the first five seconds (the entire text is generated through expanding macros), has a very simple job pagebreaking, and so forth. In addition, the percentage of letters generated through font-dependent commands is much greater in the test text than what one would find in a normal TEX manuscript. This circumstance also reduces the effect that the tabulated differences in speed will have on the overall typesetting speed for a normal TEX manuscript.

If you have not noticed that your document is being typeset slower due to the fact that the encoding used is not the encoding whose definition file was read in last, then chances are you would not notice any drop in speed if it was typeset using a relaxed encoding either.

\ActivateDefineFirst
\DeactivateDefineFirst

The DFM is turned on and off using the commands `\ActivateDefineFirst` and `\DeactivateDefineFirst`, none of which have any parameters. As it is currently implemented, the activation state of the DFM is affected by grouping, but the defining it does is global.

```
\ActivateDefineFirst
\DeactivateDefineFirst
```

3 Implementation

3.1 Initial stuff

The obligatory header code for a LATEX 2_ε package.

```
1 ⟨*package⟩
2 \NeedsTeXFormat{LaTeX2e}[1994/06/01]
3 \ProvidesPackage{relenc}[1999/01/23]
```

At the moment, there are no options for the relenc package.

3.2 Variables and similar things

\RE@temptoks

`\RE@temptoks` is used by the basic search mechaism to store the part of the search path that has not yet been used.

```
4 \newtoks\RE@temptoks
```

\RE@garbage

The `\RE@garbage` macro is used as a target for some assignments that really shouldn't be made, but where a mechanism for detecting this would not be worth the cost. It is also used as a temporary storage by some definition commands.

\RE@first@search@item

During a search, the `\RE@first@search@item` macro will expand the first item in the search path. This is needed for the define first mechanism to work.

\RE@define@first

The `\RE@define@first` macro works like a switch. If it is `\RE@active@define@first` then the define first mechanism is active: If the search does not find

anything on the first try but finds something later then the first command in the search path gets `\let` to whatever the search finds. If `\RE@define@first` is `\@gobbletwo` then the define first mechanism is inactive and nothing happens. This construction allows for other possibilities as well.

The initial setting is `\@gobbletwo`, i.e., off.

```
\let\RE@define@first\@gobbletwo
```

3.3 Search mechanisms for variable commands

Variable commands are intended to be used as definitions of encoding-specific commands or compositions of such. Thus variable commands are actually instances of some user level command, which is not the same TEX control sequence as the variable command itself.

A variable command is defined to be a parameterless macro whose expansion consists of two tokens. The first is `\RE@text@variable` if the command is not a composition and it is `\RE@text@comp@variable` if the command is a composition. The second token is the user level command itself; this command will not be expanded, its name is merely used as a word-stem from which names of possible variants are formed. Thus if the definition of the user level command `\foo` in the `T1R` encoding (which would be stored in the control sequence `\T1R\foo`) would be a variable command, that definition would be

`\RE@text@variable` `\foo`

3.3.1 Identifying the search path and initiating the search

`\RE@text@variable`
`\RE@text@comp@variable`
`\RE@spath@unavailable`
`\RE@spath@available`

The first thing `\RE@text@variable` must do is to find the search path to use. It might be either a family-specific search path, in which case it is stored in the macro `\`⟨*encoding*⟩`/`⟨*family*⟩`-path`, or the general search path for the entire encoding, in which case it is stored in the macro `\`⟨*encoding*⟩`-path`. If the family-specific search path exists, it is chosen. If neither exists, this is an error which is reported.

The second thing `\RE@text@variable` does is setting up the search along the search path. This is by expanding to

`\RE@first@read@spath` ⟨*search path*⟩ `\RE@read@spath`

which starts the next phase of the search.

```
\def\RE@text@variable{%
  \expandafter\ifx \csname\cf@encoding/\f@family-path\endcsname \relax
    \expandafter\ifx \csname\cf@encoding-path\endcsname \relax
      \RE@spath@unavailable
    \else
      \RE@spath@available\RE@first@read@spath\RE@read@spath
    \fi
  \else
    \expandafter\expandafter \expandafter\RE@first@read@spath
       \csname\cf@encoding/\f@family-path\expandafter\endcsname
       \expandafter\RE@read@spath
  \fi
}
```

Hackers may find a few things in the definition a little strange. To begin with, I use `\cf@encoding` for the name of the current encoding, although [3] says that `\f@encoding` is defined as precisely that. I too find this a little odd, but the corresponding routines in the LaTeX 2_ε kernel use `\cf@encoding` (see [1]) so I suppose that is safer, for some reason. I believe the only time at which `\cf@encoding` and `\f@encoding` are different is during a font change, but I might be mistaken about that.

The usage of `\RE@spath@unavailable` and `\RE@spath@available` might also be confusing, but the alternative to this is heavily nested `\expandafter` sequences, and those things are not easy to read. Besides, `\RE@spath@unavailable` must do something with the initial command token anyway, because `\RE@text@variable` does not move it.

`\RE@spath@available` have two real arguments: The initial search command header (which can be `\RE@first@read@spath` or `\RE@first@comp@read@spath`) and the initial search command tail (which can be `\RE@read@spath` or `\RE@comp@read@spath`). The other arguments are "dummys" used to remove unvanted tokens.

All arguments of `\RE@spath@unavailable` are such dummys.

```
\def\RE@spath@available#1#2#3\fi#4\fi{\fi\fi
   \expandafter\expandafter \expandafter#1%
      \csname\cf@encoding-path\endcsname#2%
}
\def\RE@spath@unavailable#1\fi#2\fi#3{%
   \fi\fi
   \PackageError{relenc}{%
      There is no search path for relaxed encoding \cf@encoding%
   }\@eha
}
```

Example error message:

```
! Package relenc Error: There is no search path for relaxed encoding XX.

See the relenc package documentation for explanation.
Type H <return> for immediate help
 ...

I.100


? h
Your command was ignored.
Type  I <command> <return>  to replace it with another command,
or  <return>  to continue without it.
```

The reason that `\RE@text@comp@variable` and `\RE@text@variable` exist as separate macros is that there are separate commands `\RE@first@comp@read@spath` and `\RE@first@read@spath` at the next stage of the search process.

```
\def\RE@text@comp@variable{%
  \expandafter\ifx \csname\cf@encoding/\f@family-path\endcsname \relax
    \expandafter\ifx \csname\cf@encoding-path\endcsname \relax
```

```
      \RE@spath@unavailable
    \else
      \RE@spath@available\RE@first@comp@read@spath\RE@comp@read@spath
    \fi
  \else
    \expandafter\expandafter \expandafter\RE@first@comp@read@spath
       \csname\cf@encoding/\f@family-path\expandafter\endcsname
       \expandafter\RE@comp@read@spath
  \fi
}
```

3.3.2 Performing the search

Between the steps of the second stage of the search, the initial font-dependent command (`\foo` say) will have expanded to

⟨*search header*⟩ ⟨*remaining search path*⟩ ⟨*search tail*⟩ `\foo`

The ⟨*search header*⟩ can be one of `\RE@read@spath`, `\RE@first@read@spath`, `\RE@comp@read@spath`, and `\RE@first@comp@read@spath`. The ⟨*search tail*⟩ can be either `\RE@read@spath` or `\RE@comp@read@spath`.

The ⟨*search path*⟩ consists mainly of a sequence of blocks that looks as follows

`{` ⟨*character tokens, macros*⟩ `}`

The macros should also expand to character tokens. At each step, the first of these blocks is extracted as the `#1` argument of the search header. The remaining blocks are put in the `#2` argument and if no definition is found then these will be the ⟨*remaining search path*⟩ to use when setting up for the next step. For the moment, it is put away in the token list register `\RE@temptoks`.

Argument `#1` is first checked for whether the control sequence formed by expanding

`\csname` `#1` `\string` `#3` `\endcsname`

(`#3` is the control sequence named `\foo` above) exists. If it does, it is assumed to be the definition of `\foo` for the current font, otherwise everything is set up for the next step using the ⟨*remaining search path*⟩ as search path.

There is no check to see if `#2` is empty, so if the entire search path consisted of blocks like the one above, using a font-dependent command for which no definition has been given would cause an infinite loop. There is however a hack which prevents this: When a search path is set using the proper command, the control sequence `\RE@var@not@found` is appended to the search path. Primarily, this macro is a hack that prevents the search loop from going haywire, but it also has the advantage of stopping the expansion of the search header from stripping off the braces around the last group prematurely (when

`\RE@read@spath` `{a}` `{b}` `{cd}` `\RE@read@spath` `\foo`

is expanded `#1` will be `a` and `#2` will be `{b}{cd}`, but when

`\RE@read@spath` `{b}` `{cd}` `\RE@read@spath` `\foo`

is expanded `#1` will be `b` and `#2` will be `cd`, so the next time round, `#1` will be `c` and `#2` will be `d`—the last group has been split—but if the last item in the search path is not a group, but a single token, there is no group to split).

\RE@read@spath
\RE@first@read@spath
\RE@comp@read@spath
\RE@first@comp@read@spath

Considering their usage only, the difference between `\RE@read@spath` and `\RE@comp@read@spath` is that `\RE@read@spath` is used for actual font-dependent commands while `\RE@comp@read@spath` is used for compositions of a font-dependent command and its argument. In that case, the third argument will not be `\foo` but rather something like `\\foo-a` (a single control sequence, the first backslash is the escape character and the second is part of the actual name); this particular case occurs when the user has written `\foo{a}`).

If, on the other hand, their definitions are looked at, the difference is that while `\RE@read@spath` checks if the expansion of

```
\csname#1\string#3\endcsname
```

is defined, `\RE@comp@read@spath` checks if the expansion of

```
\csname#1\@empty\string#3\endcsname
```

is defined. In most cases, these expand to the same thing (since `\@empty` is a parameterless macro which expands to nothing), but sometimes one needs to write a `#1` which gives somewhat different results depending on whether `#3` is a command or a composition. These blocks in the search path can now simply test whether the token after the parameter is `\@empty` or `\string`. The two macros that currently do this are `\RE@var@not@found` and `\RE@convert@nfss`.

The difference between `\RE@read@spath` and `\RE@first@read@spath` is that the latter is used in the first step, while the former is used in all the other steps. This difference exists because of the define first mechanism: If `\RE@first@read@spath` finds a definition in the first step, there is no need to call the mechanism no matter what the value of `\RE@define@first` is, but if `\RE@first@read@spath` does not find a definition then it must store the first block of the search path for a possible future use by `\RE@active@define@first`. `\RE@read@spath`, on the other hand, should never store a search path block, but if it finds a definition then it should call `\RE@define@first` since the define first mechanism might be active.

Quite naturally, `\RE@comp@read@spath` relates to `\RE@first@comp@read@spath` as `\RE@read@spath` to `\RE@first@read@spath` and `\RE@first@comp@read@spath` relates to `\RE@first@read@spath` as `\RE@comp@read@spath` to `\RE@read@spath` [phew!].

```
\def\RE@read@spath#1#2\RE@read@spath#3{%
  \RE@temptoks={#2}%
  \expandafter\ifx \csname#1\string#3\endcsname \relax
    \expandafter\RE@read@spath \the\expandafter\RE@temptoks
       \expandafter\RE@read@spath \expandafter#3%
  \else
    \RE@define@first{#1}{#3}%
    \csname#1\string#3\expandafter\endcsname
  \fi
}
\def\RE@first@read@spath#1#2\RE@read@spath#3{%
  \RE@temptoks={#2}%
  \expandafter\ifx \csname#1\string#3\endcsname \relax
    \def\RE@first@search@item{#1}%
    \expandafter\RE@read@spath \the\expandafter\RE@temptoks
       \expandafter\RE@read@spath \expandafter#3%
  \else
```

```
    \csname#1\string#3\expandafter\endcsname
  \fi
}
\def\RE@comp@read@spath#1#2\RE@comp@read@spath#3{%
  \RE@temptoks={#2}%
  \expandafter\ifx \csname#1\@empty\string#3\endcsname \relax
    \expandafter\RE@comp@read@spath \the\expandafter\RE@temptoks
       \expandafter\RE@comp@read@spath \expandafter#3%
  \else
    \RE@define@first{#1\@empty}{#3}%
    \csname#1\@empty\string#3\expandafter\endcsname
  \fi
}
\def\RE@first@comp@read@spath#1#2\RE@comp@read@spath#3{%
  \RE@temptoks={#2}%
  \expandafter\ifx \csname#1\@empty\string#3\endcsname \relax
    \def\RE@first@search@item{#1\@empty}%
    \expandafter\RE@comp@read@spath \the\expandafter\RE@temptoks
       \expandafter\RE@comp@read@spath \expandafter#3%
  \else
    \csname#1\@empty\string#3\expandafter\endcsname
  \fi
}
```

The definition of `\RE@read@spath` is not quite trivial. It must not leave any `\fi` up ahead, since everything should eventually expand to the actual definition of the searched-for command, and this is very likely to have parameters. There must be no surplus tokens between this command and its arguments; this is why

```
\def\RE@read@spath#1#2\RE@read@spath#3{%
  \RE@temptoks={#2}%
  \expandafter\ifx \csname#1\string#3\endcsname \relax
    \expandafter\RE@read@spath\the\RE@temptoks\RE@read@spath#3%
  \else
    \RE@define@first{#1}{#3}%
    \csname#1\string#3\endcsname
  \fi
}
```

would not do as a definition of `\RE@read@spath`.

A definition that would work is

```
\def\RE@read@spath#1#2\RE@read@spath#3{%
  \expandafter\ifx \csname#1\string#3\endcsname \relax
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi{%
    \RE@read@spath#2\RE@read@spath#3%
  }{%
    \RE@define@first{#1}{#3}%
    \csname#1\string#3\endcsname
  }%
}
```

but I prefer the "save away" definition since it makes a few additional features possible; the most important perhaps being that it gives a chance to peek in afterwards and see where the search ended (using `\ShowVariantSearchResult`).

3.3.3 Miscellaneous search-related macros

`\RE@active@define@first`
`\RE@again@read@spath`

`\RE@active@define@first` is the macro which performs the actual assignments done by the define first mechanism, if it is active. The macro simply expands to the suitable global assignment; `#1` is the block from the search path that found a definition (with `\@empty` added if the search was for a composition) and `#2` is the command or composition whose definition was searched.

```
\def\RE@active@define@first#1#2{%
   \global\expandafter\let
      \csname\RE@first@search@item\string#2\expandafter\endcsname
      \csname#1\string#2\endcsname
}
```

In some cases, the scan of the search path is resumed after a variant has been found. This is made possible by the fact that the remaining serach path is stored in `\RE@temptoks`, but the define first mechanism adds a complication. It cannot be allowed to copy a variant other than the first in the search path as that could actually change the effect of a command.

The typical case here (the only one possible to define using user level commands of relenc) is that a command has two variants, one of which is composed and one which is not. The composed occurs before the noncomposed and the default definition of the composed is to resume the scan of the search path. If the command is executed for an argument for which there is no composition, an active define first mechanism will hide all composite definitions!

Thus the define first mechanism must be temporarily disabled. One way to do this is to put a dummy value in `\RE@first@search@item`, and this solution is implemented in `\RE@again@read@spath`.

```
\def\RE@again@read@spath{%
   \def\RE@first@search@item##1\expandafter\endcsname{%
      RE@garbage\expandafter\endcsname
   }%
   \expandafter\RE@read@spath \the\RE@temptoks \RE@read@spath
}
```

`\RE@var@not@found`
`\RE@gobble@readspath`
`\RE@text@comp@unavail`

These macros put a graceful end to a search that has found nothing. When `\RE@var@not@found`, which should be the very last token in every search path (it is automatically added by `\SetEncodingSearchPath` and `\SetFamilySearchPath`), is expanded it should appear in the following context

> `\expandafter` `\ifx` `\csname` `\RE@var@not@found` ⟨*optional* `\@empty`⟩
> `\string` ⟨*command or composition*⟩ `\endcsname`

As `\RE@var@not@found` inserts an additional `\endcsname` and `\fi` when expanded by the `\csname`, the entire `\ifx` is finished before TEX gets to `\RE@gobble@readspath`. There are however a lot of mismatched tokens left over by `\RE@var@not@found`, some of which will be used in composing the error message.

```
\def\RE@var@not@found{relax\endcsname\relax\fi
   \RE@gobble@readspath
}
```

When `\RE@gobble@readspath` is expanded, it is in the context

`\RE@gobble@readspath` ⟨*optional* `\@empty`⟩ `\string`
⟨*command or composition*⟩ `\endcsname` ⟨*some text*⟩ `\fi`

The `\fi` is the `\fi` that originally matched the `\ifx` mensioned above.

The `\@empty` is there iff ⟨*command or composition*⟩ is a composition. If it is, this composition is `\string`ed and the first backslash is thrown away. Then the remaining 'other' tokens are given to `\RE@text@comp@unavail` which will figure out what is the original command and what is its argument. If it is not a composition, the command is given to `\TextSymbolUnavailable`. In any case, everything up to and including the `\fi` disappears.

```
\def\RE@gobble@readspath#1\string#2\endcsname#3\fi{%
   \ifx\@empty#1%
```

If `#2` is a composition then

```
      \expandafter\expandafter \expandafter\RE@text@comp@unavail
        \expandafter\@gobble \string#2\RE@text@comp@unavail
   \else
      \TextSymbolUnavailable{#2}%
   \fi
}
```

As there is a hyphen between the original command and its argument, `\RE@text@comp@unavail` splits the name of the composition at the first hyphen. This is the right thing to do as long as `-` is not used as part of the name of the original command (unlikely, as LaTeX does not define `\-` as an encoding-specific command).

```
\def\RE@text@comp@unavail#1-#2\RE@text@comp@unavail{%
   \PackageError{relenc}{%
      The composition of command #1 with #2\MessageBreak is declared %
      in encoding \cf@encoding,\MessageBreak but no definition could %
      be found%
   }\@eha
}
```

Example error text:

```
! Package relenc Error: The composition of command \foo with A
(relenc)                is declared in encoding XXX,
(relenc)                but no definition could be found.

See the relenc package documentation for explanation.
Type H <return> for immediate help
 ...

I.100


? h
Your command was ignored.
Type  I <command> <return>  to replace it with another command,
or  <return>  to continue without it.
```

3.4 Making variable text commands

3.4.1 Miscellaneous support macros

\RE@empty@is@qmark

Expands to its argument if that is nonempty, otherwise it expands to `?`. This macro is primarily meant to be used inside a `\csname` ... `\endcsname` block. As always, an ingenious fool might break it, but a category 3 `^^M` is extremely unlikely (and hard to type), so it should work for any reasonable argument.

```
\begingroup
   \lccode`\$=13\relax
   \lowercase{%
      \gdef\RE@empty@is@qmark#1{\ifx$#1$?\else#1\fi}%
   }
\endgroup
```

\RE@font@spec

This macro concatenates the four parameters ⟨*encoding*⟩, ⟨*family*⟩, ⟨*series*⟩, and ⟨*shape*⟩ into relenc's name for this font, i.e., puts slashes between the arguments and replaces every empty argument with a question mark.

```
\def\RE@font@spec#1#2#3#4{%
   \RE@empty@is@qmark{#1}/\RE@empty@is@qmark{#2}/%
   \RE@empty@is@qmark{#3}/\RE@empty@is@qmark{#4}%
}
```

\RE@bsl@string

One thing which complicates the definition of the definition commands (which must work even if they appear in a font definition file) is that LaTeX has the naughty habit of changing the TeX parameter `\escapechar` when it is defing a new font (loading the corresponding font definition file). The `\escapechar` parameter decides which character, if any, should be put in front of a control sequence name which is `\string`ed or written to the log file as part of some `\tracing`... operation. (To see this effect yourself, set `\tracingmacros` positive just before a font change that loads a new font definition file (better reset it to zero after the font change though) and have a look at the log file—throughout a large chunk of logged macro expansions there are no backslashes on the command names!) The reason I don't like this is that (i) it doesn't achieve anything that couldn't just as well be achieved by a combination of `\expandafter` and gobbling of unwanted tokens and (ii) some font definition files are read by LaTeX having `\escapechar` set to its normal value, so you can't trust it either way!

Still, that is how LaTeX does it and this must somehow be coped with. The `\RE@bsl@string` macro acts like `\string` but inserts a backslash in front of control sequence names even when `\string` wouldn't insert anything there. It is used instead of `\string` in all defining commands.

```
\def\RE@bsl@string{%
   \ifnum \escapechar<\z@ \@backslashchar \fi
   \string
}
```

3.4.2 Declaration commands

\DeclareTextVariableCommand

\DeclareTextVariableCommandNoDefault

\DeclareTextVariableSymbol

Two of these are the obvious counterparts of `\DeclareTextCommand` and `\DeclareTextSymbol`. The parameters are the same, and the actual declaring (from the LaTeX 2_ε kernel's point of view) is done by `\DeclareTextCommand`.

`\DeclareTextVariableCommandNoDefault` is a new possibility that exists since a relaxed encoding definition file need not actually specify the definition of any font-dependent command. It can, for example, specify that "Unless otherwise stated, use the definition from encoding ... ".

As the ordinary counterparts of these commands are preamble commands, I have made these preamble commands as well.

```
\newcommand\DeclareTextVariableCommand{\RE@dec@text@varcmd\newcommand}
\@onlypreamble\DeclareTextVariableCommand
\newcommand\DeclareTextVariableCommandNoDefault{%
   \RE@dec@text@varcmd\@gobble
}
\@onlypreamble\DeclareTextVariableCommandNoDefault
\newcommand\DeclareTextVariableSymbol[3]{%
   \RE@dec@text@varcmd\chardef#1{#2}#3\relax
}
\@onlypreamble\DeclareTextVariableSymbol
\def\RE@dec@text@varcmd#1#2#3{%
   \DeclareTextCommand{#2}{#3}{\RE@text@variable#2}%
   \expandafter#1\csname#3/?/?/?\string#2\endcsname
}
```

`\ProvideTextVariableCommand`

This is the variable counterpart of `\ProvideTextCommand`, and it has the same set of parameters. Its definition is a bit more complicated than that of the earlier commands for declaring variable font-dependent commands, partly because `\ProvideTextCommand` does not tell whether the definition it provided was used, but also becuase this command need not be meaningless even if the font-dependent command it should declare already was declared.

If the font-dependent command is not declared (the $\backslash\langle encoding\rangle\backslash\langle command\rangle$ control sequence is `\relax`), `\ProvideTextCommand` is used to declare it and `\providecommand` is used to give an encoding level definition. If the font-dependent command was declared and variable, then `\providecommand` is still called, to give an encoding-level definition of the command in case there wasn't already one. Otherwise it must have been declared as a non-variable font-dependent command and the following info message is given:

```
Package relenc Info: You have provided a declaration of \foo in
(relenc)             encoding T1R as a variable command, but it was
(relenc)             already declared as a non-variable command.
(relenc)             Your declaration has been ignored.
```

In this case, the slight problem of the left-over definition remains. This is taken care of by giving it to `\providecommand` as a definition of `\RE@garbage` (which was introduced for this kind of tasks).

`\ProvideTextCommand` is not a preamble-only command, so I have left `\ProvideTextVariableCommand` free to use anywhere too, although I cannot see why it should be needed after `\begin{document}`.

```
\newcommand\ProvideTextVariableCommand[2]{%
   \expandafter\ifx \csname#2\string#1\endcsname \relax
      \ProvideTextCommand#1#2{\RE@text@variable#1}%
      \expandafter\providecommand
         \csname#2/?/?/?\string#1\expandafter\endcsname
   \else
```

```
      \long\def\RE@garbage{\RE@text@variable#1}%
      \expandafter\ifx \csname#2\string#1\endcsname \RE@garbage
         \expandafter\providecommand
            \csname#2/?/?/?\string#1\expandafter\expandafter
            \expandafter\endcsname
      \else
         \PackageInfo{relenc}{You have provided a declaration of
             \protect#1 in\MessageBreak encoding #2 as a variable
             command, but it was\MessageBreak already declared as a
             non-variable command.\MessageBreak Your declaration has
             been ignored}%
         \expandafter\providecommand
            \csname RE@garbage\expandafter\expandafter
               \expandafter\endcsname
      \fi
   \fi
}
```

`\DeclareTextVariableAccent`

This is the variable counterpart of `\DeclareTextAccent`, and again this counterpart has the same parameters as its non-variable model. Also, it is a preamble-only command, just as its model.

```
\newcommand{\DeclareTextVariableAccent}[3]{%
   \DeclareTextCommand{#1}{#2}{\RE@text@variable#1}%
   \expandafter\newcommand \csname#2/?/?/?\string#1\endcsname
      {\add@accent{#3}}%
}
\@onlypreamble\DeclareTextVariableAccent
```

3.4.3 Definition commands

There is an important difference between the declaration commands and the definition commands, and this has to do with when they are executed. A definition command usually appears in a `.fd` file, which is read in inside a group (usually several, but that is irrelevant), but the definitions it makes should be in force after the group has ended. Therefore the definitions must be global, but the `\newcommand` family of LaTeX definition commands only make local definitions. Because of this, I have chosen to primarily use the TeX primitive `\gdef` and its relatives in the definition commands, although in several cases LaTeX-style alternatives are available as well.

`\DefineTextCommandVariant`

The basic defintion command, `\gdef` style. As simple as it can be, but note that the ⟨*parameter text*⟩ does not become one of the arguments of the macro. This has some consequences for when spaces are ignored.

```
\newcommand{\DefineTextCommandVariant}[5]{%
   \expandafter\gdef
      \csname\RE@font@spec{#2}{#3}{#4}{#5}\RE@bsl@string#1\endcsname
}
```

`\DefineTextSymbolVariant`

The basic command for defining a text symbol command.

```
\newcommand{\DefineTextSymbolVariant}[6]{%
   \global\expandafter\chardef
      \csname\RE@font@spec{#2}{#3}{#4}{#5}\RE@bsl@string#1\endcsname
      =#6\relax
```

```
}
```

\DefineTextAccentVariant

The basic command for defining an accent command. Very much a special case of \DefineTextCommandVariant.

```
\newcommand{\DefineTextAccentVariant}[6]{%
   \expandafter\gdef
      \csname\RE@font@spec{#2}{#3}{#4}{#5}\RE@bsl@string#1\endcsname
      {\add@accent{#6}}%
}
```

\NewTextCommandVariant
\RenewTextCommandVariant
\ProvideTextCommandVariant
\RE@make@text@cmd@variant
\RE@make@text@cmd@var@x

These commands are the LaTeX-style commands to define variable font-dependent commands. As the LaTeX commands they are built on is not really meant to be used for global definitions, this involves some hacking (using the LaTeX core's private control sequences to do things they are not meant to do, for example). Everything would have been much simpler if the autoload format hook \aut@global had been available in all LaTeX formats, but it is not. The purpose of \RE@make@text@cmd@variant is to save some \csname ... \endcsname pairs, the real defining is done by \RE@make@text@cmd@var@x.

```
\CheckCommand*{\newcommand}{\@star@or@long\new@command}
\newcommand{\NewTextCommandVariant}
   {\RE@make@text@cmd@variant\new@command}

\CheckCommand*{\renewcommand}{\@star@or@long\renew@command}
\newcommand{\RenewTextCommandVariant}
   {\RE@make@text@cmd@variant\renew@command}

\CheckCommand*{\providecommand}{\@star@or@long\provide@command}
\newcommand{\ProvideTextCommandVariant}
   {\RE@make@text@cmd@variant\provide@command}

\def\RE@make@text@cmd@variant#1#2#3#4#5#6{%
   \expandafter\RE@make@text@cmd@var@x
      \csname\RE@font@spec{#3}{#4}{#5}{#6}\RE@bsl@string#2\endcsname
      {#1}%
}
```

Instead of \aut@global, \l@ngrel@x is used as a hook. \l@ngrel@x is defined to be a macro whose last action is to execute \global. As \l@ngrel@x appears just before the central \def, this makes that definition global in exactly those cases where the definition is made. This requires that the topmost layer of \newcommand (and its cousins), the one in which the *-processing takes place, is stepped over. Otherwise the definition of \l@ngrel@x would be reset.

If the command has an optional argument, however, a simple

```
\def\l@ngrel@x{\global}
```

is not sufficient. In that case the command the user issues is only a preparation that inserts the default optional argument if none is present and issues the real command (the name of the real command is the name of the user command with an extra backslash prepended to it).

The real command will be globally defined, thanks to the \global at the end of \l@ngrel@x, but the preparation command is at the time of \l@ngrel@x only locally defined. This is why \l@ngrel@x \lets the preparation command to itself globally, this preserves the definition but makes it global. If the command does

not have an optional argument, this does no harm as the incorrect definition is overwritten with the correct just after `\l@ngrel@x`.

Finally, `\l@ngrel@x` resets itself to `\relax`, in case something expects it to be `\long` or `\relax`. As `\l@ngrel@x` is not reset if no command definition takes place, there is a slight chance of error in that case, but at least all LaTeX core user commands reset it before using so the hack should not have any side-effects.

```
\def\RE@make@text@cmd@var@x#1#2{%
   \def\l@ngrel@x{%
      \global\let#1#1%
      \let\l@ngrel@x\relax
      \global
   }%
   #2#1%
}
```

To examine sometime: Can `\globaldefs=1` be used instead here? As that makes all assignments global, there is a definite chance it might break something else.

3.5 Making compositions of font-dependent commands

`\RE@if@composed`

This is a general test to see if a macro is a command that is set up for having compositions, i.e., if the expansion of the macro begins with `\@text@composite`. This is the same test as is used in `\DeclareTextCompositeCommand`.

The syntax is as follows:

`\RE@if@composed` ⟨*command*⟩ `{`⟨*then-text*⟩`}{`⟨*else-text*⟩`}`

expands to ⟨*then-text*⟩ if ⟨*command*⟩ can have compositions and to ⟨*else-text*⟩ otherwise.

```
\def\RE@if@composed#1{%
   \expandafter\expandafter \expandafter\ifx
      \expandafter\@car #1\relax\relax\@nil \@text@composite
```

If it is composite then ...

```
      \expandafter\@firstoftwo
   \else
      \expandafter\@secondoftwo
   \fi
}
```

3.5.1 Commands for compositions that are variable

The LaTeX 2_ε kernel's method of forming the names of compositions is to `\string` the name of the base command and append `-`⟨*argument*⟩ to that, all of which is then `\csname`d. This generates names like

```
\\T1\foo-a
```

for the command `\foo`'s composition with the argument `a` in the `T1` encoding. As the LaTeX 2_ε kernel's mechanism for checking for compositions is used for compositions that have variable definitions as well, the names of these commands are formed using the same procedure.

Should then the names of the variants of the composition be generated from names as the above, then these names would be things like

```
\T1R/zcm/m/n\\T1R\foo-a
```

which is a bit longish, even for my taste. Therefore the names of the variants are formed from a base consisting only of the name of the font-dependent command and its argument, which is `\\foo-a` in the above example. This reduces the above to

```
\T1R/zcm/m/n\\foo-a
```

saving two to four characters. This difference in naming does however have some consequences for the usage of definitions from a non-relaxed encoding.

If, for example, `{T1}` is included in the current search path and a definition of the composition `\foo{a}` is looked for, a possible definition of this in the `T1` encoding will not be found! This is because the name tried is `\T1\\foo-a`, but the command is called `\\T1\foo-a`. This is the reason for the macro `\RE@convert@nfss`, which looks ahead to see if a composition or a plain command is currently looked for. Had instead `{\RE@convert@nfss{T1}}` been included, `\T1\\foo-a` would have been changed to `\\T1\foo-a`.

`\DeclareVariableTextComposition`
`\RE@dec@var@text@comp`

This is the variable analog of LaTeX's `\DeclareTextCompositeCommand`, although it is somewhat less as it does not make a default definition for the composition. Thus it is also an analog of `\DeclareTextVariableCommandNoDefault` for compositions.

```
\newcommand{\DeclareVariableTextComposition}[3]{%
   \expandafter\RE@dec@var@text@comp
      \csname\string#1-#3\expandafter\endcsname
      \csname\@backslashchar#2\string#1-#3\endcsname
      {#1}{#2}{#3}%
}
```

As `\DeclareVariableTextComposition` merely make a few combinations of its arguments and then call `\RE@dec@var@text@comp`, the arguments of the latter are worth an explanation. Continuing the above example of `\foo{a}` in the `T1R` encoding, the arguments will be

```
#1 is \\foo-a   #2 is \\T1R\foo-a
#3 is \foo      #4 is T1R           #5 is a
```

Should the composition already have been declared, but the definition is not variable, that definition is made the encoding level definition and the composition is made variable. If the definition is variable, an info message is given and the definition is not changed.

```
\def\RE@dec@var@text@comp#1#2#3#4#5{%
  \ifx#2\relax
```

If this composition has not been declared then ...

```
     \DeclareTextCompositeCommand{#3}{#4}{#5}%
        {\RE@text@comp@variable#1}%
  \else
     \expandafter\expandafter \expandafter\ifx
        \expandafter\@car#2\@nil
        \RE@text@variable
```

If this composition is variable then ...

```
        \PackageInfo{relenc}{Redundant \protect
           \DeclareVariableTextComposition.\MessageBreak
           The composition of \protect#3 with #5 is\MessageBreak
           already declared as a variable command\MessageBreak
           in encoding #4%
        }%
     \else
        \expandafter\let \csname#4/?/?/?\string#1\endcsname #2
        \def#2{\RE@text@comp@variable#1}%
     \fi
  \fi
}
```

Example message:

```
Package relenc Info: Redundant \DeclareVariableTextComposition.
(relenc)             The composition of \foo with a is
(relenc)             already declared as a variable command
(relenc)             in encoding T1R.
```

`\DefineTextCompositionVariant`
`\DefineTextCompositionVariantCommand`
`\RE@def@text@comp@var`

These are the commands that actually define variants of compositions. The first makes the variant a chardef token, the second makes it a parameterless macro.

The arguments for `\RE@def@text@comp@var` should be the following things: The command that does the defining (`\chardef` or `\gdef`), the base user command, the relevant encoding, family, series, and shape, and finally the argument for the composition. Note that the actual definition (slot number or replacement text, respectively) is not among the arguments of `\RE@def@text@comp@var`.

```
\newcommand\DefineTextCompositionVariant[7]{%
   \global \RE@def@text@comp@var\chardef{#1}{#2}{#3}{#4}{#5}{#6}%
      #7\relax
}

\newcommand\DefineTextCompositionVariantCommand{%
   \RE@def@text@comp@var\gdef
}

\def\RE@def@text@comp@var#1#2#3#4#5#6#7{%
   \expandafter#1%
      \csname
         \RE@font@spec{#3}{#4}{#5}{#6}\@backslashchar
         \RE@bsl@string#1-#6%
      \endcsname
}
```

`\DefineTextUncomposedVariant`
`\RE@def@text@uncmp`
`\RE@def@text@uncmp@x`

`\DefineTextUncomposedVariant` extracts the definition used for arguments that does not have definitions, inserts the given argument, and makes the resulting token sequence the definition of the variant at hand. Doing this requires that the definition of the command that the LaTeX 2_ε kernel sees is taken apart, so it might be worth describing how it is contructed.

If `\foo` is composite in the `T1` encoding, `\T1\foo` is a macro which expands to

```
\@text@composite \T1\foo #1 \@empty \@text@composite
{ ⟨noncomposite definition⟩ }
```

where ⟨*noncomposite definition*⟩ is what is needed here. The stuff before checks if the composition is defined, but in this case things are best if that part is never expanded further that this.

The actual "work" is done by `\RE@def@text@uncmp@x`, as this macro contains the `\gdef` which is the only command in this that TEX's stomach sees; everything else just expands in one way or another (unless the warning is issued, that of course contains some stomach stuff too). `\DefineTextUncomposedVariant` makes two `\csname`s: The name of the macro from which the definition is to be extracted and the name of the macro to define.

`\RE@def@text@uncmp` checks if the macro to extract from really is composed, and if it is then it expands it *one level*, feeding it the ⟨*argument*⟩ as argument. Then `\RE@def@text@uncmp@x` is given this one level expansion, and the `\@text@composite`s and everything between them is thrown away. The ⟨*noncomposite definition*⟩ is made the replacement text of a `\gdef` defining the macro to be defined. If the macro to extract a definition from is not composed however, a warning is given.

As this command refers to a specific encoding's definition of a command, the ⟨*encoding*⟩ argument mustn't be empty.

```
\newcommand\DefineTextUncomposedVariant[6]{%
   \expandafter\RE@def@text@uncmp
      \csname#2\RE@bsl@string#1\expandafter\endcsname
      \csname#2/\RE@empty@is@qmark{#3}/\RE@empty@is@qmark{#4}/%
         \RE@empty@is@qmark{#5}\@backslashchar\RE@bsl@string#1-#6%
      \endcsname
      {#6}{#1}{#2}%
}
\def\RE@def@text@uncmp#1#2#3#4#5{%
   \RE@if@composed#1{%
      \expandafter\RE@def@text@uncmp@x #1{#3}{#2}%
   }{%
      \PackageWarning{relenc}{There are no compositions for %
         \protect#4 in\MessageBreak the #5 encoding. %
         \protect\DefineTextUncomposedVariant\MessageBreak
         makes no sense here%
      }%
   }%
}
\def\RE@def@text@uncmp@x\@text@composite#1\@text@composite#2#3{%
   \gdef#3{#2}%
}
```

Example warning:

```
Package relenc Warning: There are no compositions for \foo in
(relenc)                the T1 encoding. \DefineTextUncomposedVariant
(relenc)                makes no sense here.
```

3.5.2 Commands for variants with compositions

The relenc package offers another way in which a command can be defined relative to both argument and font, and that is to allow variants of font-dependent commands to have compositions. It is pretty similar to the usual making of composite commands in LATEX.

\DefineTextVariantComposition
\DefineTextVariantCompositionCommand

These commands define the composition and make the variant a command with compositions, if it is not already. The mechanism used for compositions uses `\@text@composite`, exactly like the compositions made by commands in the LaTeX 2_ε kernel.

`\DefineTextVariantComposition` makes the final definition a chardef token. `\DefineTextVariantCompositionCommand` makes the final definition a macro without parameters.

An interesting point about these commands is that a variant need not be defined before a composition of it is made! If this happens then of course technicaly the variant gets defined, but its definition for the case when that composition is not found is simply to set things up for a continued scan of the search path. What makes this possible is that the remaining search path is stored in `\RE@temptoks`.

```
\newcommand\DefineTextVariantComposition[7]{%
   \RE@def@text@var@comp\chardef{#1}{#2}{#3}{#4}{#5}{#6}#7\relax
}
\newcommand\DefineTextVariantCompositionCommand{%
   \RE@def@text@var@comp\gdef
}
```

\RE@def@text@var@comp
\RE@make@text@comp

These are the macros that do the actual work. The parameters of `\RE@def@text@var@comp` are the user level command, the encoding, family, series, and shape in question, the argument for the composition, the assignment command that should define the composition, and finally the text that should be put after the control sequnce of the composition to complete the assignment. When the assignment command is `\chardef`, as is the case when `\DefineTextVariantComposition` makes the call, this is simply a ⟨*number*⟩—the slot. The extra `\relax` is to stop TeX from reading ahead too far when looking for the end of the ⟨*number*⟩.

The first thing `\RE@def@text@var@comp` does is to check if the variant the user wants do define a composition of is defined.

```
\def\RE@def@text@var@comp#1#2#3#4#5#6#7{%
   \expandafter\let \expandafter\RE@garbage
      \csname\RE@font@spec{#3}{#4}{#5}{#6}\RE@bsl@string#2\endcsname
   \ifx \RE@garbage\relax
```

This means the variant has not been defined. The default definition given sets up a continued scan along the search path. The key macro here is `\RE@again@read@spath` (the definition of which is found on page 24). It temporarily disables the define first mechanism and then it expands to

```
\expandafter \RE@read@spath \the \RE@temptoks \RE@read@spath
```

As the `##1` below has become `#1` when `\RE@make@text@comp` is expanded, this will act as a normal `#1` in the definition `\RE@make@text@comp` expands to.

```
      \expandafter\RE@make@text@comp\csname
         \RE@font@spec{#3}{#4}{#5}{#6}\RE@bsl@string#2%
         \endcsname {\RE@again@read@spath#2{##1}}%
```

Otherwise it must be checked if the variant is composed. If it is, then everything is fine, if it isn't, then the current definition is made the default definition.

```
   \else
      \RE@if@composed\RE@garbage{}{%
         \expandafter\RE@make@text@comp
```

```
            \csname
               \RE@font@spec{#3}{#4}{#5}{#6}\RE@bsl@string#2%
            \expandafter\endcsname
            \expandafter{\RE@garbage{##1}}%
      }%
   \fi
```

Then it is finally time to define the composition. This is pretty straight off.

```
   \global\expandafter#1\csname
      \@backslashchar\RE@font@spec{#3}{#4}{#5}{#6}\RE@bsl@string#2-#7%
   \endcsname
}
```

`\RE@make@text@comp` makes `#1` a composed command, with `#2` as the default definition. This is exactly what the corresponding command in standard LaTeX does, but for some reason LaTeX defines this command of the fly each time it is needed.

```
\def\RE@make@text@comp#1#2{%
   \gdef#1##1{\@text@composite#1##1\@empty\@text@composite{#2}}%
}
```

3.6 Miscellanenous commands

3.6.1 Search paths

A very important thing about search paths is that they are responsible for ending a search that has found nothing, as the macros which actually perform the search contain no mechanism for this. The easiest way to ensure this is to append the token `\RE@var@not@found` to every search path set, and this is exactly what the standard commands for setting search paths do. Anyone who does not use these commands for setting a search path must be aware that if a search path does not contain any block that ends the search then some tiny errors are likely to start infinite loops.

`\SetEncodingSearchPath`
`\SetFamilySearchPath`
`\RE@set@spath`
`\RE@spath@catcodes`

These are all fairly standard. `\RE@set@spath` does the actual setting of a search path while `\SetEncodingSearchPath` and `\SetFamilySearchPath` merely form the name of the macro that will store the search path and temporarily change some `\catcode`s to make typing the search path somewhat easier. The `\catcode`s are changed by `\RE@spath@catcodes`.

```
\newcommand{\SetEncodingSearchPath}[1]{%
   \begingroup
   \RE@spath@catcodes
   \expandafter\RE@set@spath \csname#1-path\endcsname
}
\newcommand{\SetFamilySearchPath}[2]{%
   \begingroup
   \RE@spath@catcodes
   \expandafter\RE@set@spath \csname#1/#2-path\endcsname
}
\def\RE@set@spath#1#2{%
   \gdef#1{#2\RE@var@not@found}%
   \endgroup
}
```

```
\def\RE@spath@catcodes{%
   \catcode`\ =9\relax
   \catcode`\^^I=9\relax
   \catcode`\^^M=9\relax
   \catcode`\@=11\relax
   \catcode`\/=12\relax
   \catcode`\?=12\relax
}
```

`\RE@convert@nfss`

For some relaxed encodings, one wants the definition of a font-dependent command to be identical to the definition of the command in a non-relaxed encoding (this is the case with `T1R`, which is designed to be a relaxed version of `T1`). As there are slight differences in how the names of compositions are generated between the LaTeX 2_ε kernel and relenc, it would not be sufficient to simply include `{T1}` in the search path of the `T1R` encoding to make it use `T1`'s definitions of compositions as variants of that composition. The purpose of `\RE@convert@nfss` is to overcome this by changing the names of the variants of compositions where required. Including `\RE@convert@nfss{T1}` in the search path of `T1R` gives the intended effect.

```
\def\RE@convert@nfss#1#2{%
   \ifx\@empty#2%
      \@backslashchar#1\expandafter\expandafter \expandafter\@gobble
   \else
      #1\expandafter#2%
   \fi
}
```

3.6.2 The define first mechanism

`\ActivateDefineFirst`
`\DeactivateDefineFirst`

These commands can be used to turn the define first mechanism on and off. I am not sure at the moment whether implementing it was a good idea, but as it exists it might as well stay so it can be evaluated.

It could be worth pointing out that these commands act locally. The define first mechanism, on the other hand, acts globally.

```
\newcommand\ActivateDefineFirst{%
   \let\RE@define@first\RE@active@define@first
}
\newcommand\DeactivateDefineFirst{%
   \let\RE@define@first\@gobbletwo
}
```

3.7 Debugging

`\ShowVariantSearchResult`

This command offers a way to peek into the search mechanism, and in particular to see where it stopped. This command has a potential to come in real handy, but note that it is the *remaining search path* that is shown. You do not see the block that was actually used, you see all blocks that come after it.

```
\newcommand{\ShowVariantSearchResult}{%
   \immediate\write\sixt@@n{Encoding: \cf@encoding}%
   \immediate\write\sixt@@n{Family:   \f@family}%
   \immediate\write\sixt@@n{Series:   \f@series}%
   \immediate\write\sixt@@n{Shape:    \f@shape}%
```

```
    \immediate\write\sixt@@n
       {Remaining search path:\MessageBreak\the\RE@temptoks}%
    \show\RE@first@search@item
}
```

3.8 Fix for a bug in LaTeX

The definition of the control sequence `\@text@composite@x` in LaTeXes released 1998 and earlier has a bug (number 2930 in the LaTeX bugs database). The definition in question is

```
\def\@text@composite@x#1#2{%
   \ifx#1\relax
      \expandafter#2%
   \else
      #1%
   \fi}
```

The problem with this definition is the `\expandafter`. According to [1], it was "added so that less chaos ensues from the misuse of this code with commands that take several arguments", but unfortunately, it was placed in a position where it can do no good. `#2` is in all reasonable cases a sequence of more than one token, hence the `\expandafter` will not expand the following `\else`, but some token in `#2` instead! I suspect that the `\expandafter` was really meant to be placed in front of the `#1`, which actually is just a single token.

As it happens, the above bug can cause some serious problems when the non-composite definition of a composite command happens to be variable, since the above `\expandafter` will then expand a control sequence which should *not* be expanded.

`\@text@composite@x`

The following code fixes the above bug if the LaTeX used is one with that bug. This is done as a curtesy to all those LaTeX users which (like myself) are not always updating LaTeX as soon as a new release appears. The definition of `\@text@composite@x` is checked and, if it matches the above, replaced with the fixed definition.

```
\def\RE@garbage#1#2{%
   \ifx#1\relax
      \expandafter#2%
   \else
      #1%
   \fi}
\ifx \@text@composite@x\RE@garbage
   \def\@text@composite@x#1{%
      \ifx #1\relax
        \expandafter\@secondoftwo
      \else
        \expandafter\@firstoftwo
      \fi
      #1%
   }
\fi
\let\RE@garbage\relax
⟨/package⟩
```

The T1R encoding

A Specification

The T1R encoding is meant to be (almost) equivalent to the T1 encoding from the author's point of view but offer larger flexibility to font designers than the T1 encoding does—especially the flexibility to include additional ligatures. It is thus what I call a *relaxed* version of the T1 encoding, which is also the reason for its name.

Its basic derivation from T1 can be easily described: Slots 0–63, 64–127, and 192–255 have exactly the same contents as in the T1 encoding, whilst slots 128–191 can be used in any way the font designer wants; the set of commands, syntactic ligatures, and directly usable characters is exactly the same as in the T1 encoding. The result of my intention to give a formal description of the encoding might of course be that this description deviates slightly from the above, but it lines out the basic ideas.

The reasons for choosing this basis are pretty arbitrary, especially in terms of the assignment of slots. To begin with it gives a simple rule (the 256 slots in the font are divided into four continuous blocks of 64 slots each) and secondly almost all glyphs in slot ranges 0–63, 64–127, and 192–255 can be found in many popular fonts, but most of the glyphs in the slot range 128–191 are hard to find. Hence these glyphs will very frequently have to be made somehow anyway, the question is only at what time this will happen: At the time TEX is running, or afterwards, when a virtual font is interpreted? The character usually looks the same.

That is not the same thing as saying that it does not matter which, because it may well do. TEX will not automatically hyphenate words containing letters that are not a single character in the font, so some care is necessary. It is however the case that documents that both use and require hyphenation of all the accented characters that appear in slots 128–191 are extremely few (or do no exist at all—yet), so there are usually plenty of slots available for the font designer to put, for example, additional ligatures in. It might however well be that two different implementations of the same font (presumably made on different locations) have kept different sets of characters, due to the fact that the two implementors speak different languages and hence need hyphentation of different sets of accented letters. This causes no problems (unless the implementors are exchanging dvi files), as the code in the font definition files would mirror this difference and set up the variable commands correctly anyhow, so that exchanging .tex manuscripts does not cause problems.

Now for the more formal description, with reservation for that (i) I might have missed some point about the T1 encoding, lacking a formal definition of that and (ii) the current version of the T1R encoding is only a beta, so some details may well change in the future. In particular, the decisions about whether a composition should be a composition of a variant or a composition of the main variable command were usually pretty randomly made, so if someone should present me with a good reason why it should not be as it currently is, then there is a good chance I would change it[11].

[11] Unless too many people's code have already come to depend on it. But in that case, there is always the possibility to make new encoding that differs from T1R only in a few such points and recommend people to use that instead.

A.1 The coding scheme

Every coding scheme for a `T1R` encoded font should comply to what is specified in Tables 2, 3, and 4. The corresponding table for slots 128–191 would read "not specified" in every entry in the glyph columns, so it has been omitted. 'not specified' means that the contents of that slot is not specified, so it is completely up to the font designer to decide, although there is always a default glyph even for the unassigned slots (viz. the same glyph as in the `T1` encoding). In most cases, there must be some font-dependent command variant definitions in the font definition file for each slot that deviates from the default.

The author is guaranteed to be able to access the characters in slots 32–126 simply through character tokens.

A.2 The syntactic ligatures

The required syntactic ligatures in the `T1R` encoding are listed in Table 5. Actually, the '`-`' (*hyphen*) + *hyphenchar* $\mapsto$ *hyphenchar* ligature is a kind of "semi-aestetic" ligature, as it might have an aestetic function as well as a syntactic. When TEX hyphenates a word, a *hyphenchar* is automatically inserted at the hyphenation point, but when the hyphenation point falls immediately after an explicit hyphen, this would result in two hyphens in the printed output, if it was not for this ligature. If you know what you do, you may change or even leave this ligature out, but be prepared that such an action must be attuned to, amongst other things, the hyphenation patterns and the current value of the TEX font parameter `\hyphenchar`.

A.3 The font dimensions

As for the required font dimensions, I have chosen to only require the seven font dimensions that all TEX fonts have in common. These are listed in Table 6. As both the `ec` family of fonts and the `T1` encoded fonts made by *fontinst* (as of v 1.8, see [2]) have seven more however, it seems like that set of fourteen font dimensions should be considered the current `T1` standard. I will probably have added the other seven by the time I get to the first non-beta release, but I have refrained from including them in this release since I do not feel sure enough about what they are to write a formal specification for them. It should be noted, though, that the specification only lists a minimal set of font dimensions—therefore including the other seven in a font is most likely only good.

A.4 The font-dependent commands

The font-dependent commands of the `T1R` encoding fall into two categories: symbol commands and accenting commands. The symbol commands simply typeset one symbol. The accenting commands take one argument and typesets the result of accenting the material that the argument would typeset.

A list of the symbol commands of the `T1R` encoding can be found in Table 7. One thing worth noticing about this table is that it lists some glyphs, for example *sterling*, that are not listed in the required coding scheme, yet the font designer is required to provide the author with these. Normally this would be done by simply including the glyph in the font in the same slot as in the `T1` encoding, but if a

Slot			Glyph	Slot			Glyph
Oct.	Dec.	Hex.		Oct.	Dec.	Hex.	
0	0	00	‘`’ (*grave accent*)	40	32	20	‘␣’ (*visible space*)
1	1	01	‘´’ (*acute accent*)	41	33	21	‘!’
2	2	02	‘^’ (*circumflex accent*)	42	34	22	‘"’ (*quotedbl*)
3	3	03	‘~’ (*tilde accent*)	43	35	23	‘#’
4	4	04	‘¨’ (*dieresis*)	44	36	24	‘$’
5	5	05	*Hungarian umlaut*	45	37	25	‘%’
6	6	06	‘˚’ (*ring accent*)	46	38	26	‘&’
7	7	07	‘ˇ’ (*caron accent*)	47	39	27	‘’’ (*quoteright*)
10	8	08	‘˘’ (*breve accent*)	50	40	28	‘(’
11	9	09	‘¯’ (*macron accent*)	51	41	29	‘)’
12	10	0A	*dot accent*	52	42	2A	‘*’ (*asterisk*)
13	11	0B	‘¸’ (*cedilla accent*)	53	43	2B	‘+’
14	12	0C	*ogonek accent*	54	44	2C	‘,’ (*comma*)
15	13	0D	*quotesinglbase*	55	45	2D	‘-’ (*hyphen*)
16	14	0E	*guilsinglleft*	56	46	2E	‘.’
17	15	0F	*guilsinglright*	57	47	2F	‘/’ (*slash*)
20	16	10	*quotedblleft*	60	48	30	‘0’ (*zero*)
21	17	11	*quotedblright*	61	49	31	‘1’
22	18	12	*quotedblbase*	62	50	32	‘2’
23	19	13	*guillemotleft*	63	51	33	‘3’
24	20	14	*guillemotright*	64	52	34	‘4’
25	21	15	*endash*	65	53	35	‘5’
26	22	16	*emdash*	66	54	36	‘6’
27	23	17	*compwordmark*	67	55	37	‘7’
30	24	18	*perthousandzero*	70	56	38	‘8’
31	25	19	‘ı’ (*dotlessi*)	71	57	39	‘9’
32	26	1A	‘ȷ’ (*dotlessj*)	72	58	3A	‘:’
33	27	1B	not specified	73	59	3B	‘;’
34	28	1C	not specified	74	60	3C	‘<’
35	29	1D	not specified	75	61	3D	‘=’
36	30	1E	not specified	76	62	3E	‘>’
37	31	1F	not specified	77	63	3F	‘?’

Table 2: The coding scheme for the T1R encoding, slots 0–63

Slot			Glyph	Slot			Glyph
Oct.	Dec.	Hex.		Oct.	Dec.	Hex.	
10	64	40	‘@’	140	96	60	‘‘’ (*quoteleft*)
11	65	41	‘A’	141	97	61	‘a’
12	66	42	‘B’	142	98	62	‘b’
13	67	43	‘C’	143	99	63	‘c’
14	68	44	‘D’	144	100	64	‘d’
15	69	45	‘E’	145	101	65	‘e’
16	70	46	‘F’	146	102	66	‘f’
17	71	47	‘G’	147	103	67	‘g’
110	72	48	‘H’	150	104	68	‘h’
111	73	49	‘I’	151	105	69	‘i’
112	74	4A	‘J’	152	106	6A	‘j’
113	75	4B	‘K’	153	107	6B	‘k’
114	76	4C	‘L’	154	108	6C	‘l’
115	77	4D	‘M’	155	109	6D	‘m’
116	78	4E	‘N’	156	110	6E	‘n’
117	79	4F	‘O’	157	111	6F	‘o’
120	80	50	‘P’	160	112	70	‘p’
121	81	51	‘Q’	161	113	71	‘q’
122	82	52	‘R’	162	114	72	‘r’
123	83	53	‘S’	163	115	73	‘s’
124	84	54	‘T’	164	116	74	‘t’
125	85	55	‘U’	165	117	75	‘u’
126	86	56	‘V’	166	118	76	‘v’
127	87	57	‘W’	167	119	77	‘w’
130	88	58	‘X’	170	120	78	‘x’
131	89	59	‘Y’	171	121	79	‘y’
132	90	5A	‘Z’	172	122	7A	‘z’
133	91	5B	‘[’	173	123	7B	‘{’
134	92	5C	‘\’	174	124	7C	‘|’
135	93	5D	‘]’	175	125	7D	‘}’
136	94	5E	‘^’ (*circumflex character*)	176	126	7E	‘~’ (*tilde character*)
137	95	5F	‘_’	177	127	7F	*hyphenchar*

Table 3: The coding scheme for the T1R encoding, slots 64–127

Slot			Glyph	Slot			Glyph
Oct.	Dec.	Hex.		Oct.	Dec.	Hex.	
30	192	C0	'À'	340	224	E0	'à'
31	193	C1	'Á'	341	225	E1	'á'
32	194	C2	'Â'	342	226	E2	'â'
33	195	C3	'Ã'	343	227	E3	'ã'
34	196	C4	'Ä'	344	228	E4	'ä'
35	197	C5	'Å'	345	229	E5	'å'
36	198	C6	'Æ'	346	230	E6	'æ'
37	199	C7	'Ç'	347	231	E7	'ç'
310	200	C8	'È'	350	232	E8	'è'
311	201	C9	'É'	351	233	E9	'é'
312	202	CA	'Ê'	352	234	EA	'ê'
313	203	CB	'Ë'	353	235	EB	'ë'
314	204	CC	'Ì'	354	236	EC	'ì'
315	205	CD	'Í'	355	237	ED	'í'
316	206	CE	'Î'	356	238	EE	'î'
317	207	CF	'Ï'	357	239	EF	'ï'
320	208	D0	*Eth*	360	240	F0	*eth*
321	209	D1	'Ñ'	361	241	F1	'ñ'
322	210	D2	'Ò'	362	242	F2	'ò'
323	211	D3	'Ó'	363	243	F3	'ó'
324	212	D4	'Ô'	364	244	F4	'ô'
325	213	D5	'Õ'	365	245	F5	'õ'
326	214	D6	'Ö'	366	246	F6	'ö'
327	215	D7	'Œ' (*OE*)	367	247	F7	'œ'
330	216	D8	'Ø'	370	248	F8	'ø'
331	217	D9	'Ù'	371	249	F9	'ù'
332	218	DA	'Ú'	372	250	FA	'ú'
333	219	DB	'Û'	373	251	FB	'û'
334	220	DC	'Ü'	374	252	FC	'ü'
335	221	DD	'Ý'	375	253	FD	'ý'
336	222	DE	*Thorn*	376	254	FE	*thorn*
337	223	DF	*SS*	377	255	FF	'ß' (*germandbls*)

Table 4: The coding scheme for the T1R encoding, slots 192–255

'-' (*hyphen*)	+	'-' (*hyphen*)	↦	*endash*
endash	+	'-' (*hyphen*)	↦	*emdash*
'!'	+	'¿'	↦	'¡'
','	+	','	↦	*quotedblright*
','	+	','	↦	*quotedblbase*
'-' (*hyphen*)	+	*hyphenchar*	↦	*hyphenchar*
'<'	+	'<'	↦	*guillemotleft*
'>'	+	'>'	↦	*guillemotright*
'?'	+	'¿'	↦	'¿'
'¿'	+	'¿'	↦	*quotedblleft*

Table 5: The required syntactic ligatures in the T1R encoding

No.	Meaning	No.	Meaning
1	Slant per pt	5	x-height (size of 1 ex)
2	Interword space	6	Quad width (size of 1 em)
3	Interword stretch	7	Extra space
4	Interword shrink		

Table 6: Required \fontdimens in an T1R encoded font

large number of slots must be used for other glyphs, such as ligatures, then this may not be possible. In such cases the symbol command could instead typeset a symbol in *another* font. This could be done with a definition such as

```
\DefineTextCommandVariant{\textsterling}{T1R}{zcm}{}{}%
  {\UseTextSymbol{T1}{\textsterling}}
```

In order to make this particular piece of code work, the font designer would of course need to set up an entire family of T1-encoded fonts in parallell with the main T1R-encoded font family. Alternatively, one could instead collect all these miscellaneous glyphs from an entire family in one font (that would probably become U-encoded), but then one needs to define a lot more variants. In any case, this trick of implementing a command as using a glyph in another font cannot be used for '¡' and '¿' , as these need to be accessible through syntactic ligatures as well. What's more, font designers to be should be aware that one cannot have kerns between two glyphs in different fonts.

A list of the accenting commands of the T1R encoding can be found in Table 8. It is hardly exciting, but the font designer should find the information in the 'Remark' column interesting; it lists all the compositions of the accenting commands and whether they are implemented as compositions of the main command or compositions of some variant.

A.5 On hyphenation patterns for the T1R encoding

TEX is constructed so that the hyphenation patterns must match the coding scheme of the font used for the text that is to be hyphenated. It is therefore possible that hyphenation patterns made for use with the T1 encoding does not always work with the T1R encoding. They will work, however, with a font whose

Command	Action	Remark
\AE	Typeset ‘Æ’	
\ae	Typeset ‘æ’	
\DH	Typeset *Eth*	
\dh	Typeset *eth*	
\DJ	Typeset *Eth*	
\dj	Typeset *dbar*	Variable
\guillemotleft	Typeset *guillemotleft*	
\guillemotright	Typeset *guillemotright*	
\guilsinglleft	Typeset *guilsinglleft*	
\guilsinglright	Typeset *guilsinglright*	
\i	Typeset ‘ı’ (*dotlessi*)	
\j	Typeset ‘ȷ’ (*dotlessj*)	
\L	Typeset *Lslash*	Variable
\l	Typeset *lslash*	Variable
\NG	Typeset *Eng*	Variable
\ng	Typeset *eng*	Variable
\OE	Typeset ‘Œ’ (*OE*)	
\oe	Typeset ‘œ’	
\O	Typeset ‘Ø’	
\o	Typeset ‘ø’	
\quotedblbase	Typeset *quotedblbase*	
\quotesinglbase	Typeset *quotesinglbase*	
\SS	Typeset *SS*	Variable
\ss	Typeset ‘ß’	
\textasciicircum	Typeset ‘^’ (*circumflex character*)	
\textasciitilde	Typeset ‘~’ (*tilde character*)	
\textbackslash	Typeset ‘\’	
\textbar	Typeset ‘|’	
\textbraceleft	Typeset ‘{’	
\textbraceright	Typeset ‘}’	
\textcompwordmark	Typeset *compwordmark*	
\textdollar	Typeset ‘$’	
\textemdash	Typeset *emdash*	
\textendash	Typeset *endash*	
\textexclamdown	Typeset ‘¡’	Variable
\textgreater	Typeset ‘>’	
\textless	Typeset ‘<’	
\textperthousand	Typeset ‘%’ followed by one *perthousandzero*	
\textpertenthousand	Typeset ‘%’ followed by two *perthousandzero*	
\textquestiondown	Typeset ‘¿’	Variable
\textquotedblleft	Typeset *quotedblleft*	
\textquotedblright	Typeset *quotedblright*	
\textquotedbl	Typeset ‘"’ (*quotedbl*)	
\textquoteleft	Typeset ‘‘’	
\textquoteright	Typeset ‘’’	
\textsection	Typeset *section*	Variable
\textsterling	Typeset *sterling*	Variable
\textunderscore	Typeset ‘_’	
\textvisiblespace	Typeset ‘␣’	
\TH	Typeset *Thorn*	
\th	Typeset *thorn*	

Table 7: The symbol commands of the T1R encoding

Command	Action	Remark
\‘	Typeset the argument with a grave accent above it	Compositions for A, a, E, e, I, i, \i, O, o, U, and u
\’	Typeset the argument with an acute accent above it	Compositions for A, a, C (var.), c (var.), E, e, I, i, \i, L (var.), l (var.), N (var.), n (var.), O, o, R (var.), r (var.), S (var.), s (var.), U, u, Y, y, Z (var.), and z (var.)
\^	Typeset the argument with a circumflex accent above it	Compositions for A, a, E, e, I, i, \i, O, o, U, and u
\~	Typeset the argument with a tilde accent above it	Compositions for A, a, N, n, O, and o
\"	Typeset the argument with a dieresis accent above it	Compositions for A, a, E, e, I, i, \i, O, o, U, u, Y (var.), and y (var.)
\H	Typeset the argument with a Hungarian umlaut above it	Variable; compositions of the encoding level variant for O, o, U, and u
\r	Typeset the argument with a ring accent above it	Compositions for A, a, U (var.), and u (var.)
\v	Typeset the argument with a caron accent above it	Variable; compositions of the encoding level variant for C, c, D, d, E, e, L, l, N, n, R, r, S, s, T, t, Z, and z
\u	Typeset the argument with a breve accent above it	Variable; compositions of the encoding level variant for A, a, G, and g
\=	Typeset the argument with a macron accent above it	
\.	Typeset the argument with a dot accent above it	Variable; compositions of the encoding level variant for I, i, Z, and z
\b	Typeset the argument with a macron accent under it	
\c	Typeset the argument with a cedilla accent under it	Variable; compositions for C, c, S (var.), s (var.), T (var.), and t (var.)
\d	Typeset the argument with a dot accent under it	
\k	Typeset the argument with an ogonek accent under it	Variable; compositions of the encoding level variant for A, a, E, and e

Table 8: The accenting commands of the T1R encoding

coding scheme deviates from that of the T1 encoding only in that some ligatures (or possibly some symbols that never occur as part of a word) has replaced some letters. This is because when TEX hyphenates a word, it treats a ligature as the sequence of letters it is composed from and not as the single character it may be in the font. It does not matter whether there is a hyphenation pattern mensioning character n or not when character n occurs in the output only as a ligature of other characters.

Even in other cases, there is a good chance that hyphenation patterns made for use with the T1 encoding will work without modification for a font with the T1R encoding. The main reason for this is that one usually does not have hyphenation patterns involving letters of which one does not intend to form automatically hyphenatable words active. If one wants hyphenation of words containing a certain character, one also wants it to have a slot of its own in the coding scheme of the font. In most cases, this means that the character is in the same slot as in the T1 encoding.

B Implementation

First the file announces itself.

```
⟨*encoding⟩
\ProvidesFile{t1renc.def}
   [1998/12/17 Relaxed TeX latin text encoding, version 1.00 (beta)]
```

Then there is a check for if the relenc package has been read. If it hasn't, there is no point in contiuing.

```
\@ifundefined{RE@text@variable}{%
   \PackageError{T1R encoding}{%
      The definition of the T1R encoding requires that\MessageBreak
      the 'relenc' package is loaded first}%
   {The T1R encoding cannot be defined.\MessageBreak
      If you continue, you will most likely face further errors.%
      \MessageBreak The best option is to type 'x' and fix your
      manuscript.}%
   \endinput
}{}
```

The encoding declares itself and its font substitution.

```
\DeclareFontEncoding{T1R}{}{}
\DeclareFontSubstitution{T1R}{zcm}{m}{n}
```

The search path is set; this is something only relaxed encodings do. The most noteworthy point about this search path is that it includes searching for definitions from the T1 encoding. Thanks to this, the T1R encoding can do without definitions of its own for many variable commands (saving some memory), so the lines defining these have been commented out below.

```
\SetEncodingSearchPath{T1R}{
   {\cf@encoding/\f@family/\f@series/\f@shape}
   {\cf@encoding/\f@family/?/\f@shape}
   {\cf@encoding/\f@family/?/?}
   {\cf@encoding/?/?/?}
   {\RE@convert@nfss{T1}}
   {\RE@convert@nfss{?}}
```

```
}
```

The accenting commands are declared. I have choosen to use the same default definitions as in the `T1` encoding, despite the fact that the definitions used for `\c` and `\k` do not always do what they should (accents are positioned in curious places). These commands are declared to be variable though, so a font designer can override them with ones that are better suited for the font family in question.

```
\DeclareTextAccent{\‘}{T1R}{0}
\DeclareTextAccent{\’}{T1R}{1}
\DeclareTextAccent{\^}{T1R}{2}
\DeclareTextAccent{\~}{T1R}{3}
\DeclareTextAccent{\"}{T1R}{4}
\DeclareTextVariableAccent{\H}{T1R}{5}
\DeclareTextAccent{\r}{T1R}{6}
\DeclareTextVariableAccent{\v}{T1R}{7}
\DeclareTextVariableAccent{\u}{T1R}{8}
\DeclareTextAccent{\=}{T1R}{9}
\DeclareTextVariableAccent{\.}{T1R}{10}
\DeclareTextCommand{\b}{T1R}[1]
   {{\o@lign{\relax#1\crcr\hidewidth\sh@ft{29}%
     \vbox to.2ex{\hbox{\char9}\vss}\hidewidth}}}
\DeclareTextVariableCommand{\c}{T1R}[1]
   {\setbox\z@\hbox{#1}\ifdim\ht\z@=1ex\accent11 #1%
     \else{\ooalign{\hidewidth\char11\hidewidth
        \crcr\unhbox\z@}}\fi}
\DeclareTextCommand{\d}{T1R}[1]
   {{\o@lign{\relax#1\crcr\hidewidth\sh@ft{10}.\hidewidth}}}
\DeclareTextVariableCommand{\k}{T1R}[1]
   {\oalign{\null#1\crcr\hidewidth\char12}}
```

The symbol commands are declared. These are pretty straightforward, but *SS* is a bit of a special case. The command `\SS` must exist as the upper case equivalent of `\ss`, but is there any case where the *SS* glyph is different from two S's?[12]

```
\DeclareTextSymbol{\AE}{T1R}{198}
\DeclareTextSymbol{\DH}{T1R}{208}
\DeclareTextSymbol{\DJ}{T1R}{208}
\DeclareTextVariableSymbol{\L}{T1R}{138}
\DeclareTextVariableSymbol{\NG}{T1R}{141}
\DeclareTextSymbol{\OE}{T1R}{215}
\DeclareTextSymbol{\O}{T1R}{216}
\DeclareTextVariableSymbol{\SS}{T1R}{223}
\DeclareTextSymbol{\TH}{T1R}{222}
\DeclareTextSymbol{\ae}{T1R}{230}
\DeclareTextSymbol{\dh}{T1R}{240}
\DeclareTextVariableSymbol{\dj}{T1R}{158}
\DeclareTextSymbol{\guillemotleft}{T1R}{19}
\DeclareTextSymbol{\guillemotright}{T1R}{20}
\DeclareTextSymbol{\guilsinglleft}{T1R}{14}
\DeclareTextSymbol{\guilsinglright}{T1R}{15}
\DeclareTextSymbol{\i}{T1R}{25}
\DeclareTextSymbol{\j}{T1R}{26}
```

[12] I am currently thinking about removing the *SS* glyph from the required coding scheme, so if anyone has any opinions on this particular matter, please share them with me.

```
\DeclareTextVariableSymbol{\l}{T1R}{170}
\DeclareTextVariableSymbol{\ng}{T1R}{173}
\DeclareTextSymbol{\oe}{T1R}{247}
\DeclareTextSymbol{\o}{T1R}{248}
\DeclareTextSymbol{\quotedblbase}{T1R}{18}
\DeclareTextSymbol{\quotesinglbase}{T1R}{13}
\DeclareTextSymbol{\ss}{T1R}{255}
\DeclareTextSymbol{\textasciicircum}{T1R}{`\^}
\DeclareTextSymbol{\textasciitilde}{T1R}{`\~}
\DeclareTextSymbol{\textbackslash}{T1R}{`\\}
\DeclareTextSymbol{\textbar}{T1R}{`\|}
\DeclareTextSymbol{\textbraceleft}{T1R}{`\{}
\DeclareTextSymbol{\textbraceright}{T1R}{`\}}
\DeclareTextSymbol{\textcompwordmark}{T1R}{23}
\DeclareTextSymbol{\textdollar}{T1R}{`\$}
\DeclareTextSymbol{\textemdash}{T1R}{22}
\DeclareTextSymbol{\textendash}{T1R}{21}
\DeclareTextVariableSymbol{\textexclamdown}{T1R}{189}
\DeclareTextSymbol{\textgreater}{T1R}{`\>}
\DeclareTextSymbol{\textless}{T1R}{`\<}
\DeclareTextCommand{\textperthousand}{T1R}{\%\char 24 }
\DeclareTextCommand{\textpertenthousand}{T1R}{\%\char 24\char 24 }
\DeclareTextVariableSymbol{\textquestiondown}{T1R}{190}
\DeclareTextSymbol{\textquotedblleft}{T1R}{16}
\DeclareTextSymbol{\textquotedblright}{T1R}{17}
\DeclareTextSymbol{\textquotedbl}{T1R}{`\"}
\DeclareTextSymbol{\textquoteleft}{T1R}{`\`}
\DeclareTextSymbol{\textquoteright}{T1R}{`\'}
\DeclareTextVariableSymbol{\textsection}{T1R}{159}
\DeclareTextVariableSymbol{\textsterling}{T1R}{191}
\DeclareTextSymbol{\textunderscore}{T1R}{95}
\DeclareTextSymbol{\textvisiblespace}{T1R}{32}
\DeclareTextSymbol{\th}{T1R}{254}
```

The last thing to declare are all the compositions. It starts with the compositions from block 2 (slots 128–191), which are all in some way variable.

128 = "80.

```
\DefineTextVariantComposition{\.}{T1R}{}{}{}{i}{`\i}
\DefineTextVariantComposition{\u}{T1R}{}{}{}{A}{128}
\DefineTextVariantComposition{\k}{T1R}{}{}{}{A}{129}
\DeclareVariableTextComposition{\'}{T1R}{C}
% \DefineTextCompositionVariant{\'}{T1R}{}{}{}{C}{130}
\DefineTextVariantComposition{\v}{T1R}{}{}{}{C}{131}
\DefineTextVariantComposition{\v}{T1R}{}{}{}{D}{132}
\DefineTextVariantComposition{\v}{T1R}{}{}{}{E}{133}
\DefineTextVariantComposition{\k}{T1R}{}{}{}{E}{134}
\DefineTextVariantComposition{\u}{T1R}{}{}{}{G}{135}
```

136 = "88.

```
\DeclareVariableTextComposition{\'}{T1R}{L}
% \DefineTextCompositionVariant{\'}{T1R}{}{}{}{L}{136}
\DefineTextVariantComposition{\v}{T1R}{}{}{}{L}{137}
\DeclareVariableTextComposition{\'}{T1R}{N}
% \DefineTextCompositionVariant{\'}{T1R}{}{}{}{N}{139}
\DefineTextVariantComposition{\v}{T1R}{}{}{}{N}{140}
```

```
\DefineTextVariantComposition{\H}{T1R}{}{}{}{O}{142}
\DeclareVariableTextComposition{\'}{T1R}{R}
% \DefineTextCompositionVariant{\'}{T1R}{}{}{}{R}{143}
```

144 = "90.

```
\DefineTextVariantComposition{\v}{T1R}{}{}{}{R}{144}
\DeclareVariableTextComposition{\'}{T1R}{S}
% \DefineTextCompositionVariant{\'}{T1R}{}{}{}{S}{145}
\DefineTextVariantComposition{\v}{T1R}{}{}{}{S}{146}
\DeclareVariableTextComposition{\c}{T1R}{S}
% \DefineTextCompositionVariant{\c}{T1R}{}{}{}{S}{147}
\DefineTextVariantComposition{\v}{T1R}{}{}{}{T}{148}
\DeclareVariableTextComposition{\c}{T1R}{T}
% \DefineTextCompositionVariant{\c}{T1R}{}{}{}{T}{149}
\DefineTextVariantComposition{\H}{T1R}{}{}{}{U}{150}
\DeclareVariableTextComposition{\r}{T1R}{U}
% \DefineTextCompositionVariant{\r}{T1R}{}{}{}{U}{151}
```

152 = "98.

```
\DeclareVariableTextComposition{\"}{T1R}{Y}
% \DefineTextCompositionVariant{\"}{T1R}{}{}{}{Y}{152}
\DeclareVariableTextComposition{\'}{T1R}{Z}
% \DefineTextCompositionVariant{\'}{T1R}{}{}{}{Z}{153}
\DefineTextVariantComposition{\v}{T1R}{}{}{}{Z}{154}
\DefineTextVariantComposition{\.}{T1R}{}{}{}{Z}{155}
\DefineTextVariantComposition{\.}{T1R}{}{}{}{I}{157}
```

160 = "A0.

```
\DefineTextVariantComposition{\u}{T1R}{}{}{}{a}{160}
\DefineTextVariantComposition{\k}{T1R}{}{}{}{a}{161}
\DeclareVariableTextComposition{\'}{T1R}{c}
% \DefineTextCompositionVariant{\'}{T1R}{}{}{}{c}{162}
\DefineTextVariantComposition{\v}{T1R}{}{}{}{c}{163}
\DefineTextVariantComposition{\v}{T1R}{}{}{}{d}{164}
\DefineTextVariantComposition{\v}{T1R}{}{}{}{e}{165}
\DefineTextVariantComposition{\k}{T1R}{}{}{}{e}{166}
\DefineTextVariantComposition{\u}{T1R}{}{}{}{g}{167}
```

168 = "A8.

```
\DeclareVariableTextComposition{\'}{T1R}{l}
% \DefineTextCompositionVariant{\'}{T1R}{}{}{}{l}{168}
\DefineTextVariantComposition{\v}{T1R}{}{}{}{l}{169}
\DeclareVariableTextComposition{\'}{T1R}{n}
% \DefineTextCompositionVariant{\'}{T1R}{}{}{}{n}{171}
\DefineTextVariantComposition{\v}{T1R}{}{}{}{n}{172}
\DefineTextVariantComposition{\H}{T1R}{}{}{}{o}{174}
\DeclareVariableTextComposition{\'}{T1R}{r}
% \DefineTextCompositionVariant{\'}{T1R}{}{}{}{r}{175}
```

176 = "B0.

```
\DefineTextVariantComposition{\v}{T1R}{}{}{}{r}{176}
\DeclareVariableTextComposition{\'}{T1R}{s}
% \DefineTextCompositionVariant{\'}{T1R}{}{}{}{s}{177}
\DefineTextVariantComposition{\v}{T1R}{}{}{}{s}{178}
\DeclareVariableTextComposition{\c}{T1R}{s}
% \DefineTextCompositionVariant{\c}{T1R}{}{}{}{s}{179}
```

```
\DefineTextVariantComposition{\v}{T1R}{}{}{}{t}{180}
\DeclareVariableTextComposition{\c}{T1R}{t}
% \DefineTextCompositionVariant{\c}{T1R}{}{}{}{t}{181}
\DefineTextVariantComposition{\H}{T1R}{}{}{}{u}{182}
\DeclareVariableTextComposition{\r}{T1R}{u}
% \DefineTextCompositionVariant{\r}{T1R}{}{}{}{u}{183}
```

184 = "B8.

```
\DeclareVariableTextComposition{\"}{T1R}{y}
% \DefineTextCompositionVariant{\"}{T1R}{}{}{}}{y}{184}
\DeclareVariableTextComposition{\'}{T1R}{z}
% \DefineTextCompositionVariant{\'}{T1R}{}{}{}}{z}{185}
\DefineTextVariantComposition{\v}{T1R}{}{}{}{z}{186}
\DefineTextVariantComposition{\.}{T1R}{}{}{}{z}{187}
```

Here follows the compositions in block 3 (slots 192–255). They are exactly the same as in the `T1` encoding.

192 = "C0.

```
\DeclareTextComposite{\`}{T1R}{A}{192}
\DeclareTextComposite{\'}{T1R}{A}{193}
\DeclareTextComposite{\^}{T1R}{A}{194}
\DeclareTextComposite{\~}{T1R}{A}{195}
\DeclareTextComposite{\"}{T1R}{A}{196}
\DeclareTextComposite{\r}{T1R}{A}{197}
\DeclareTextComposite{\c}{T1R}{C}{199}
```

200 = "C8.

```
\DeclareTextComposite{\`}{T1R}{E}{200}
\DeclareTextComposite{\'}{T1R}{E}{201}
\DeclareTextComposite{\^}{T1R}{E}{202}
\DeclareTextComposite{\"}{T1R}{E}{203}
\DeclareTextComposite{\`}{T1R}{I}{204}
\DeclareTextComposite{\'}{T1R}{I}{205}
\DeclareTextComposite{\^}{T1R}{I}{206}
\DeclareTextComposite{\"}{T1R}{I}{207}
```

208 = "D0.

```
\DeclareTextComposite{\~}{T1R}{N}{209}
\DeclareTextComposite{\`}{T1R}{O}{210}
\DeclareTextComposite{\'}{T1R}{O}{211}
\DeclareTextComposite{\^}{T1R}{O}{212}
\DeclareTextComposite{\~}{T1R}{O}{213}
\DeclareTextComposite{\"}{T1R}{O}{214}
```

216 = "D8.

```
\DeclareTextComposite{\`}{T1R}{U}{217}
\DeclareTextComposite{\'}{T1R}{U}{218}
\DeclareTextComposite{\^}{T1R}{U}{219}
\DeclareTextComposite{\"}{T1R}{U}{220}
\DeclareTextComposite{\'}{T1R}{Y}{221}
```

224 = "E0.

```
\DeclareTextComposite{\`}{T1R}{a}{224}
\DeclareTextComposite{\'}{T1R}{a}{225}
\DeclareTextComposite{\^}{T1R}{a}{226}
\DeclareTextComposite{\~}{T1R}{a}{227}
```

```
\DeclareTextComposite{\"}{T1R}{a}{228}
\DeclareTextComposite{\r}{T1R}{a}{229}
\DeclareTextComposite{\c}{T1R}{c}{231}
```

232 = "E8.

```
\DeclareTextComposite{\‘}{T1R}{e}{232}
\DeclareTextComposite{\’}{T1R}{e}{233}
\DeclareTextComposite{\^}{T1R}{e}{234}
\DeclareTextComposite{\"}{T1R}{e}{235}
\DeclareTextComposite{\‘}{T1R}{i}{236}
\DeclareTextComposite{\‘}{T1R}{\i}{236}
\DeclareTextComposite{\’}{T1R}{i}{237}
\DeclareTextComposite{\’}{T1R}{\i}{237}
\DeclareTextComposite{\^}{T1R}{i}{238}
\DeclareTextComposite{\^}{T1R}{\i}{238}
\DeclareTextComposite{\"}{T1R}{i}{239}
\DeclareTextComposite{\"}{T1R}{\i}{239}
```

240 = "F0.

```
\DeclareTextComposite{\~}{T1R}{n}{241}
\DeclareTextComposite{\‘}{T1R}{o}{242}
\DeclareTextComposite{\’}{T1R}{o}{243}
\DeclareTextComposite{\^}{T1R}{o}{244}
\DeclareTextComposite{\~}{T1R}{o}{245}
\DeclareTextComposite{\"}{T1R}{o}{246}
```

248 = "F8.

```
\DeclareTextComposite{\‘}{T1R}{u}{249}
\DeclareTextComposite{\’}{T1R}{u}{250}
\DeclareTextComposite{\^}{T1R}{u}{251}
\DeclareTextComposite{\"}{T1R}{u}{252}
\DeclareTextComposite{\’}{T1R}{y}{253}
```

⟨/encoding⟩

The zcm example font family

C The zcm font family and reldemo.tex

The zcm family of fonts consists of only two fonts and it is meant to accompany the relenc package documentation. These fonts are really just combinations of glyphs taken from other fonts, primarily the Computer Modern fonts (hence the cm), but there are also some glyphs which are taken from some LaTeX symbol fonts (lasy10 and lcircle10).

The primary reason these fonts exist is that I felt I needed an example of what can be done with the relenc package. As one really cannot expect that people will have any font that is not either a Computer Modern or a LaTeX font, such an example has to be made using these fonts. I am well aware that some of the glyphs look terrible, but the zcm family is not intended to be used for any serious typesetting.

The font family consists of two fonts: zcmr8d and zcmra. The first of these is declared as T1R/zcm/m/n and its coding scheme is identical to that of the T1

encoding. The second is declared as T1R/zcm/m/a and its coding scheme deviates from that of the T1 encoding in four slots, which have been reassigned to contain ligatures instead of accented letters.

A demonstration of these fonts is generated by typesetting reldemo.tex; this document contains a font table listing the contents of each slot, a demonstration of all the symbol commands, a demonstration of the effects of applying the accenting commands to the non-accented letters, and a comprehensive list of character pairs, for both fonts. The reason I have put it in a separate file is (i) that the relenc package will have to be installed before these fonts can be viewed (the files needed are relenc.sty, t1renc.def, and t1rzcm.fd) and (ii) that there might be some problems connected to viewing these fonts.

To begin with, both fonts are virtual fonts and some DVI drivers cannot handle these. If you only have such drivers, you should get one that can (not just for the sake of the relenc package—virtual fonts are in general neat to be able to use).

One of the fonts used to make these virtaul fonts—the lcircle10 font—occurs under two other names as well: circle10 and lcirc10. Unfortunately, one cannot expect that there is more than one of these in any given TEX installation. Thus you will get some problems if your installation uses another name for this font. Some DVI drivers will translate the name used in the virtual font to that of the font names which actually occurs in the system, and in this case there is no problem, but if not then you will have to intervene yourself.

The easiest way to do this is simply to rename a few virtual font files, since there are such files accompanying the relenc package for using the lcircle10 font under any one of these names. What you have to see too is that the ones which use the font under the name on your system are the ones which are named zcmr8d.vf and zcmra.vf respectively. The following table shows what the file names are and which font it uses

Virtual font file	lcircle10 name used
zcmr8d.vf	lcricle10
zcmra.vf	lcricle10
zcmr8d.vf2	cricle10
zcmra.vf2	cricle10
zcmr8d.vf3	lcric10
zcmra.vf3	lcric10

Note that all the above files are virtual font files. The reason only two have been given the correct suffix .vf is that there really are no more than two fonts, so they should not take up any more font name space than that either.

Finally, there is one other use for the T1R/zcm/m/n font—it is the final substitution font in the T1R encoding. LATEX requires that there is such a font, so it might as well be this one. This means that you should see to that the files which are this font (zcmr8d.tfm and zcmr8d.vf) are put in such locations that TEX and its helper programs (DVI drivers etc.) will find them if they are needed. You should not need to move them before you typeset reldemo.tex for the first time, however, as these fonts are initially in the same directory as this and TEX looks for files there first.

If you do have the ec family of fonts however, you may (quite understandably) want to use one of them as final substitution font for the T1R encoding instead. There is a package called ecsubzcm that is installed with relenc. This package

declares the zcm family in encoding T1R and ecrm1000 under the encoding/family/series/shape combination T1R/zcm/m/n. After loading this package, LaTeX will not bother to input the font definition file t1rzcm.fd, so the "real" zcm fonts will not get involved.

D Implementation

D.1 Font definition file

As is the custom, the file starts by announcing itself.

```
592 ⟨*fd⟩
593 \ProvidesFile{t1rzcm.fd}[1999/01/19 Font definitions for T1R/zcm.]
```

Then the family is declared. So far, there is nothing out of the ordinary.

```
594 \DeclareFontFamily{T1R}{zcm}{}
```

Next the shapes are declared. There is nothing strange about this either.

```
595 \DeclareFontShape{T1R}{zcm}{m}{n}{
596    <-> zcmr8d
597 }{}
598 \DeclareFontShape{T1R}{zcm}{m}{a}{
599    <-> zcmra
600 }{}
```

That would be all in a normal font definition file. This file does however go on with defining some variants of commands and compositions, to compensate for the differences between the coding scheme of the zcmra font and the default coding scheme.

Slot number 128 is used for a ligature instead of the default 'Ă', so this has to be compensated for. In this case it is done by making another default definition of the \u command and defining all the other compositions of that command again.

```
601 \DefineTextAccentVariant{\u}{T1R}{zcm}{}{a}{8}
602 \DefineTextVariantComposition{\u}{T1R}{zcm}{}{a}{G}{135}
603 \DefineTextVariantComposition{\u}{T1R}{zcm}{}{a}{a}{160}
604 \DefineTextVariantComposition{\u}{T1R}{zcm}{}{a}{g}{167}
```

It should be noted at this point that this font definition file is not set up in the most efficient way. Normally, one would want to minimize the number of variants that needs to be defined—something which the above most certainly does not do—and one often achieves this by trying to affect as few accenting commands as possible. The four extra ligatures of the zcmra font could have been put in slots 128, 135, 160, and 167—that would have made it possible to get by with the single variant definition

```
\DefineTextAccentVariant{\u}{T1R}{zcm}{}{a}{8}
```

although a font designer who frequently use the \u command will probably prefer to use slots for compositions of some other command—but zcmra has been set up more to show what is possible than what is optimal.

The same problem occurs with slot 131, which does not contain the 'Č' glyph that the encoding-level composition of \v with C assumes it does. Another solution to the problem is to give an explicit definition of the composition at a lower level, as in

```
\DefineTextVariantCompositionCommand{\v}{T1R}{zcm}{}{a}{C}{%
    \add@accent{7}{C}%
}
```

`\add@accent` is standard LaTeX and its definition can be found in [1]. This solution gets by using only two variants (the `T1R/zcm/?/a` variant of `\v`, and the compositon of that variant with `C`). Had instead the first solution been used here, it would have required the code

```
\DefineTextAccentVariant{\v}{T1R}{zcm}{}{a}{7}
\DefineTextVariantComposition{\v}{T1R}{zcm}{}{a}{D}{132}
\DefineTextVariantComposition{\v}{T1R}{zcm}{}{a}{E}{133}
\DefineTextVariantComposition{\v}{T1R}{zcm}{}{a}{L}{137}
\DefineTextVariantComposition{\v}{T1R}{zcm}{}{a}{N}{140}
\DefineTextVariantComposition{\v}{T1R}{zcm}{}{a}{R}{144}
\DefineTextVariantComposition{\v}{T1R}{zcm}{}{a}{S}{146}
\DefineTextVariantComposition{\v}{T1R}{zcm}{}{a}{T}{148}
\DefineTextVariantComposition{\v}{T1R}{zcm}{}{a}{Z}{154}
\DefineTextVariantComposition{\v}{T1R}{zcm}{}{a}{c}{163}
\DefineTextVariantComposition{\v}{T1R}{zcm}{}{a}{d}{164}
\DefineTextVariantComposition{\v}{T1R}{zcm}{}{a}{e}{165}
\DefineTextVariantComposition{\v}{T1R}{zcm}{}{a}{l}{169}
\DefineTextVariantComposition{\v}{T1R}{zcm}{}{a}{n}{172}
\DefineTextVariantComposition{\v}{T1R}{zcm}{}{a}{r}{176}
\DefineTextVariantComposition{\v}{T1R}{zcm}{}{a}{s}{178}
\DefineTextVariantComposition{\v}{T1R}{zcm}{}{a}{t}{180}
\DefineTextVariantComposition{\v}{T1R}{zcm}{}{a}{z}{186}
```

which defines another 18 control sequences. The `\v` command sure has many compositions, hasn't it?

Things are a bit different when the slot is used for a variant of a composition, instead of a composition of a variant. In the former case, there is a special command that can be used to extract the default definition of the accenting command.

```
\DefineTextUncomposedVariant{\'}{T1R}{zcm}{}{a}{C}
```

Finally, another example like `\v`.

```
\DefineTextVariantCompositionCommand{\k}{T1R}{zcm}{}{a}{a}%
    {a\llap{\char12\kern-0.07em}}
```

⟨/fd⟩

This definition demonstrates that there is absolutely no need to base the definition of a composition on the default definition; using that in this case would (i) be much longer, (ii) be unstable, as the default definition of `\k` positions the accent incorrectly if not both the height of the accent and the depth of the letter is zero or less, and (iii) probably not kern particularly good (at least this definition kerns like 'a' to the left, the default does not kern at all).

D.2 The **ecsubzcm** package

⟨∗package⟩

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{ecsubzcm}[1999/01/19]
\DeclareFontFamily{T1R}{zcm}{}
\DeclareFontShape{T1R}{zcm}{m}{n}{
```

```
    <-> ecrm1000
}{}
⟨/package⟩
```

References

[1] Johannes Braams, David Carlisle, Alan Jeffrey, Frank Mittelbach, Chris Rowley, Rainer Schöpf: `ltoutenc.dtx` (part of the LaTeX 2ε base distribution).

[2] Alan Jeffrey, Rowland McDonnell (manual), Sebastian Rahtz, Ulrik Vieth: *The fontinst utility* (v 1.8), `fontinst.dtx`, in CTAN at `ftp://ftp.tex.ac.uk/tex-archive/fonts/utilities/fontinst/`...

[3] LaTeX3 Project Team: *LaTeX 2ε font selection*, `fntguide.tex` (part of the LaTeX 2ε base distribution).

[4] Frank Mittelbach [et al. ?]: `encguide.tex`. To appear as part of the LaTeX 2ε base distribution. Sometime. Or at least, that is the intention.