

# Babel

Version 3.9g  
2013/10/10

*Original author*  
Johannes L. Braams

*Current maintainer*  
Javier Bezos

The standard distribution of  $\LaTeX$  contains a number of document classes that are meant to be used, but also serve as examples for other users to create their own document classes. These document classes have become very popular among  $\LaTeX$  users. But it should be kept in mind that they were designed for American tastes and typography. At one time they even contained a number of hard-wired texts.

This manual describes babel, a package that makes use of the capabilities of  $\TeX$  version 3 and, to some extent, xetex and luatex, to provide an environment in which documents can be typeset in a language other than US English, or in more than one language or script. However, no attempt has been done to take full advantage of the features provided by the latter, which would require a completely new core (as for example polyglossia or as part of  $\LaTeX 3$ ).

# Contents

<b>1</b>	<b>The user interface</b>	<b>3</b>
1.1	Selecting languages . . . . .	4
1.2	Getting the current language name . . . . .	5
1.3	Shorthands . . . . .	6
1.4	Package options . . . . .	9
1.5	The base option . . . . .	11
1.6	Hooks . . . . .	11
1.7	Hyphenation tools . . . . .	12
1.8	Language attributes . . . . .	14
1.9	Languages supported by babel . . . . .	14
1.10	Tips, workarounds, know issues and notes . . . . .	15
1.11	Future work . . . . .	16
<b>2</b>	<b>Preloading languages with language.dat</b>	<b>17</b>
<b>3</b>	<b>The interface between the core of babel and the language definition files</b>	<b>18</b>
3.1	Basic macros . . . . .	19
3.2	Skeleton . . . . .	20
3.3	Support for active characters . . . . .	21
3.4	Support for saving macro definitions . . . . .	22
3.5	Support for extending macros . . . . .	22
3.6	Macros common to a number of languages . . . . .	22
3.7	Encoding-dependent strings . . . . .	23
<b>4</b>	<b>Compatibility and changes</b>	<b>27</b>
4.1	Compatibility with german.sty . . . . .	27
4.2	Compatibility with ngerman.sty . . . . .	27
4.3	Compatibility with the french package . . . . .	27
4.4	Changes in babel version 3.9 . . . . .	27
4.5	Changes in babel version 3.7 . . . . .	27
4.6	Changes in babel version 3.6 . . . . .	29
4.7	Changes in babel version 3.5 . . . . .	30
<b>5</b>	<b>Identification and loading of required files</b>	<b>30</b>
5.1	Multiple languages . . . . .	31
<b>6</b>	<b>The Package File</b>	<b>32</b>
6.1	base . . . . .	33
6.2	key=value options and other general option . . . . .	33
6.3	Conditional loading of shorthands . . . . .	35
6.4	Language options . . . . .	36
<b>7</b>	<b>The Kernel of Babel</b>	<b>39</b>
7.1	Tools . . . . .	40
7.2	Encoding issues . . . . .	41
7.3	Support for active characters . . . . .	44
7.4	Shorthands . . . . .	45
7.5	Conditional loading of shorthands . . . . .	52
7.6	Language attributes . . . . .	53
7.7	Support for saving macro definitions . . . . .	56

7.8	Support for extending macros . . . . .	57
7.9	Hyphens . . . . .	57
7.10	Macros common to a number of languages . . . . .	59
7.11	Making glyphs available . . . . .	59
7.12	Quotation marks . . . . .	60
7.13	Letters . . . . .	61
7.14	Shorthands for quotation marks . . . . .	62
7.15	Umlauts and trema's . . . . .	63
7.16	Multienencoding strings . . . . .	64
7.17	Hooks . . . . .	69
7.18	The redefinition of the style commands . . . . .	70
7.19	Cross referencing macros . . . . .	70
7.20	Marks . . . . .	74
7.21	Preventing clashes with other packages . . . . .	75
7.21.1	ifthen . . . . .	75
7.21.2	varioref . . . . .	76
7.21.3	hhline . . . . .	76
7.21.4	hyperref . . . . .	77
7.21.5	fancyhdr . . . . .	77
7.22	Encoding issues (part 2) . . . . .	78
7.23	Local Language Configuration . . . . .	78
7.23.1	Redefinition of macros . . . . .	80
7.24	Multiple languages . . . . .	83
<b>8</b>	<b>The 'nil' language</b>	<b>97</b>
<b>9</b>	<b>Support for Plain T<sub>E</sub>X</b>	<b>98</b>
9.1	Not renaming hyphen.tex . . . . .	98
9.2	Emulating some L <sup>A</sup> T <sub>E</sub> X features . . . . .	99
<b>10</b>	<b>Tentative font handling</b>	<b>106</b>
<b>11</b>	<b>Hooks for XeT<sub>E</sub>X and LuaT<sub>E</sub>X</b>	<b>106</b>
11.1	XeT <sub>E</sub> X . . . . .	106
11.2	LuaT <sub>E</sub> X . . . . .	107
<b>12</b>	<b>Conclusion</b>	<b>108</b>
<b>13</b>	<b>Acknowledgements</b>	<b>108</b>

# 1 The user interface

The basic user interface of this package is quite simple. It consists of a set of commands that switch from one language to another, and a set of commands that deal with shorthands. It is also possible to find out what the current language is. In most cases, a single language is required, and then all you need in L<sup>A</sup>T<sub>E</sub>X is to load the package using its standard mechanism for this purpose, namely, passing that language as an optional argument. In multilingual documents, just use several options. So, in L<sup>A</sup>T<sub>E</sub>X2<sub>ε</sub> the preamble of the document:

```
\documentclass{article}
\usepackage[dutch,english]{babel}
```

would tell L<sup>A</sup>T<sub>E</sub>X that the document would be written in two languages, Dutch and English, and that English would be the first language in use, and the main one. You can also set the main language explicitly:

```
\documentclass{article}
\usepackage[main=english,dutch]{babel}
```

Another approach is making dutch and english global options in order to let other packages detect and use them:

```
\documentclass[dutch,english]{article}
\usepackage{babel}
\usepackage{varioref}
```

In this last example, the package `varioref` will also see the options and will be able to use them.

Languages may be set as global and as package option at the same time, but in such a case you should set explicitly the main language with the package option `main`:

```
\documentclass[italian]{babel}
\usepackage[ngerman,main=italian]{babel}
```

**New 3.9c** The basic behaviour of some languages can be modified when loading `babel` by means of *modifiers*. They are set after the language name, and are prefixed with a dot (only when the language is set as package option – neither global options nor the `main` key accept them). An example is (spaces are not significant and they can be added or removed):<sup>1</sup>

```
\usepackage[latin.medieval, spanish.notilde.lcroman, danish]{babel}
```

Attributes (described below) are considered modifiers, ie, you can set an attribute by including it in the list of modifiers. However, modifiers is a more general mechanism.

Currently `babel` provides no standard interface for scripts. Languages sharing the same non-Latin script may define macros to switch them (eg, `\textcyrillic`), but be aware they may also set the language to a certain default. (Future versions might add such an interface.)

Because of the way `babel` has evolved, “language” can refer to (1) a set of hyphenation patterns as preloaded into the format, (2) a package option, (3) an `ldf`

<sup>1</sup>No predefined “axis” for modifiers are provided because languages and their scripts have quite different needs.

file, and (4) a name used in the document to select a language or dialect. So, a package option refers to a language in a generic way – sometimes it is the actual language name used to select it, sometimes it is a file name loading a language with a different name, sometimes it is a file name loading several languages. Please, read the documentation for specific languages for further info. Loading directly sty files in L<sup>A</sup>T<sub>E</sub>X (ie, `\usepackage{<language>}`) is deprecated and you will get the error:<sup>2</sup>

```
! Package babel Error: You are loading directly a language style.
(babel)                This syntax is deprecated and you must use
(babel)                \usepackage[language]{babel}.
```

In Plain, load languages styles with `\input` and then use `\begindocument` (the latter is defined by babel):

```
\input estonian.sty
\begindocument
```

Note not all languages provide a sty file and some of them are not compatible with Plain.

## 1.1 Selecting languages

This section describes the commands to be used in the document to switch the language in multilingual document.

The main language is selected automatically when the document environment begins. In the preamble it has *not* been selected, except hyphenation patterns and the name assigned to `\language`. If you need to define boxes and the like in the preamble, you might want to use some of the following commands.

`\selectlanguage` {<language>}

When a user wants to switch from one language to another he can do so using the macro `\selectlanguage`. This macro takes the language, defined previously by a language definition file, as its argument. It calls several macros that should be defined in the language definition files to activate the special definitions for the language chosen. For “historical reasons”, a macro name is converted to a language name without the leading `\`; in other words, the two following declarations are equivalent:

```
\selectlanguage{german}
\selectlanguage{\german}
```

Using a macro instead of a “real” name is deprecated.

If used inside braces there might be some non-local changes, as this would be roughly equivalent to:

```
{\selectlanguage{<inner-language>} ...}\selectlanguage{<outer-language>}
```

If you want a change which is really local, you must enclose this code with an additional grouping level.

This command can be used as environment, too.

`\begin{otherlanguage}` {<language>} ... `\end{otherlanguage}`

The environment `otherlanguage` does basically the same as `\selectlanguage`, except the language change is (mostly) local to the environment. Actually, there might be some non-local changes, as this environment is roughly equivalent to:

```
\begingroup
\selectlanguage{<inner-language>}
...
\endgroup
\selectlanguage{<outer-language>}
```

If you want a change which is really local, you must enclose this environment with an additional grouping, like braces `{}`. Spaces after the environment are ignored.

`\foreignlanguage` {<language>}{<text>}

The command `\foreignlanguage` takes two arguments; the second argument is a phrase to be typeset according to the rules of the language named in its first argument. This command (1) only switches the extra definitions and the hyphenation rules for the language, *not* the names and dates, (2) does not send information about the language to auxiliary files (i.e., the surrounding language is still in force), and (3) it works even if the language has not been set as package option (but in such a case it only sets the hyphenation patterns).

`\begin{otherlanguage*}` {<language>} ... `\end{otherlanguage*}`

Same as `\foreignlanguage` but as environment. Spaces after the environment are *not* ignored.

This environment (or in some cases `otherlanguage`) may be required for intermixing left-to-right typesetting with right-to-left typesetting in engines not supporting a change in the writing direction inside a line.

`\begin{hyphenrules}` {<language>} ... `\end{hyphenrules}`

The environment `hyphenrules` can be used to select *only* the hyphenation rules to be used (it can be used as command, too). This can for instance be used to select ‘nohyphenation’, provided that in `language.dat` the ‘language’ nohyphenation is defined by loading `zerohyph.tex`. It deactivates language shorthands, too (but not user shorthands).

Except for these simple uses, `hyphenrules` is discouraged and `otherlanguage*` (the starred version) is preferred, as the former does not take into account possible changes in encodings or characters like, say, ‘ ’ done by some languages (eg, `italian`, `frenchb`, `ukraineb`). To set hyphenation exceptions, use `\babelhyphenation` (see below).

## 1.2 Getting the current language name

`\language` The control sequence `\language` contains the name of the current language. However, due to some internal inconsistencies in catcodes it should *not* be used to test its value (use `iflang`, by Heiko Oberdiek).

<sup>2</sup>In former versions the error read “You have used an old interface to call babel”, not very helpful.

`\iflanguage`  $\{\langle language \rangle\}\{\langle true \rangle\}\{\langle false \rangle\}$

If more than one language is used, it might be necessary to know which language is active at a specific time. This can be checked by a call to `\iflanguage`, but note here “language” is used in the T<sub>E</sub>X sense, as a set of hyphenation patterns, and *not* as its babel name. This macro takes three arguments. The first argument is the name of a language; the second and third arguments are the actions to take if the result of the test is true or false respectively. The advice about `\language` also applies here – use `iflang` instead of `\iflanguage` if possible.

### 1.3 Shorthands

A *shorthand* is a sequence of one or two characters that expands to arbitrary T<sub>E</sub>X code.

Shorthands can be used for different kinds of things, as for example: (1) in some languages shorthands such as “a are defined to be able to hyphenate the word if the encoding is OT1; (2) in some languages shorthands such as ! are used to insert the right amount of white space; (3) several kinds of discretionaries and breaks can be inserted easily with “-, “=, etc.

The package `inputenc` as well as `xetex` and `luatex` have alleviated entering non-ASCII characters, but minority languages and some kinds of text can still require characters not directly available in the keyboards (and sometimes not even as separated or precomposed Unicode characters). As to the point 2, now pdfT<sub>E</sub>X provides `\knbcode`. Tools of point 3 can be still very useful in general.

There are three levels of shorthands: *user*, *language*, and *system* (by order of precedence). Version 3.9 introduces the *language user* level on top of the user level, as described below. In most cases, you will use only shorthands provided by languages.

Please, note the following:

1. Activated chars used for two-char shorthands cannot be followed by a closing brace `}` and the spaces following are gobbled. With one-char shorthands (eg, `:`), they are preserved.
2. If on a certain level (system, language, user) there is a one-char shorthand, two-char ones starting with that char and on the same level are ignored.

A typical error when using shorthands is the following:

! Argument of `\language@active@arg` has an extra `}`.

It means there is a closing brace just after a shorthand, which is not allowed (eg, `"}`). Just add `{}` after (eg, `"{}}`).

`\shorthandon`  $\{\langle shorthands-list \rangle\}$   
`\shorthandoff`  $*\{\langle shorthands-list \rangle\}$

It is sometimes necessary to switch a shorthand character off temporarily, because it must be used in an entirely different way. For this purpose, the user commands `\shorthandoff` and `\shorthandon` are provided. They each take a list of characters as their arguments.

The command `\shorthandoff` sets the `\catcode` for each of the characters in its argument to other (12); the command `\shorthandon` sets the `\catcode` to active (13). Both commands only work on ‘known’ shorthand characters. If a character is not known to be a shorthand character its category code will be left unchanged.

**New 3.9** Note however, `\shorthandoff` does not behave as you would expect with characters like `~` or `^`, because they usually are not “other”. For them `\shorthandoff*` is provided, so that with

```
\shorthandoff*{~^}
```

`~` is still active, very likely with the meaning of a non-breaking space, and `^` is the superscript character. The catcodes used are those when the shorthands are defined, usually when language files are loaded.

**`\usesshorthands`** `*{\langle char \rangle}`

The command `\usesshorthands` initiates the definition of user-defined shorthand sequences. It has one argument, the character that starts these personal shorthands.

**New 3.9** However, user shorthands are not always alive, as they may be deactivated by languages (for example, if you use `"` for your user shorthands and switch from german to french, they stop working). Therefore, a starred version `\usesshorthands*{\langle char \rangle}` is provided, which makes sure shorthands are always activated.

Currently, if the package option `shorthands` is used, you must include any character to be activated with `\usesshorthands`. This restriction will be lifted in a future release.

**`\defineshorthand`** `[\langle language \rangle, \langle language \rangle, \dots] \{\langle shorthand \rangle\} \{\langle code \rangle\}`

The command `\defineshorthand` takes two arguments: the first is a one- or two-character shorthand sequence, and the second is the code the shorthand should expand to.

**New 3.9** An optional argument allows to (re)define language and system shorthands (some languages do not activate shorthands, so you may want to add `\languageshorthands{\langle lang \rangle}` to the corresponding `\extras{\langle lang \rangle}`). By default, user shorthands are (re)defined.

User shorthands override language ones, which in turn override system shorthands. Language-dependent user shorthands (new in 3.9) take precedence over “normal” user shorthands.

As an example of their applications, let’s assume you want a unified set of shorthand for dictionaries (languages do not define shorthands consistently, and `"`, `\`, `"` have different meanings). You could start with, say:

```
\usesshorthands*{"}
\defineshorthand{"*}{\babelhyphen{soft}}
\defineshorthand{"-}{\babelhyphen{hard}}
```

However, behaviour of hyphens is language dependent. For example, in languages like Polish and Portugese, a hard hyphen inside compound words are repeated at the beginning of the next line. You could then set:

```
\defineshorthand[*polish,*portugese]{"-}{\babelhyphen{repeat}}
```

Here, options with `*` set a language-dependent user shorthand, which means the generic one above only applies for the rest of languages; without `*` they would (re)define the language shorthands instead, which are overridden by user ones. Now, you have a single unified shorthand (`"` -), with a content-based meaning (‘compound word hyphen’) whose visual behavior is that expected in each context.



`\aliasshorthand`  $\{\langle original \rangle\}\{\langle alias \rangle\}$

The command `\aliasshorthand` can be used to let another character perform the same functions as the default shorthand character. If one prefers for example to use the character `/` over `"` in typing Polish texts, this can be achieved by entering `\aliasshorthand{/}{/}`. *Please note* the substitute character must *not* have been declared before as shorthand (in such case, `\aliasshorthands` is ignored). The following example shows how to replace a shorthand by another

```
\aliasshorthand{~}{^}
\AtBeginDocument{\shorthandoff*{~}}
```

However, shorthands remember somehow the original character, and the fallback value is that of the latter. So, in this example, if no shorthand is found, `^` expands to a non-breaking space, because this is the value of `~` (internally, `^` calls `\active@char~` or `\normal@char~`). Furthermore, if you change the system value of `^` with `\defineshorthand` nothing happens.

`\languageshorthands`  $\{\langle language \rangle\}$

The command `\languageshorthands` can be used to switch the shorthands on the language level. It takes one argument, the name of a language or none (the latter does what its name suggests).<sup>3</sup> Note that for this to work the language should have been specified as an option when loading the babel package. For example, you can use in english the shorthands defined by `ngerman` with

```
\addto\extrasinglish{\languageshorthands{ngerman}}
```

Very often, this is a more convenient way to deactivate shorthands than `\shorthandoff`, as for example if you want to define a macro to easy typing phonetic characters with `tipa`:

```
\newcommand{\myipa}[1]{\{\{\languageshorthands{none}\tipaencoding#1\}}
```

`\babelshorthand`  $\{\langle shorthand \rangle\}$

With this command you can use a shorthand even if (1) not activated in shorthands (in this case only shorthands for the current language are taken into account, ie, not user shorthands), (2) turned off with `\shorthandoff` or (3) deactivated with the internal `\bbl@deactivate`; for example, `\babelshorthand{"u}` or `\babelshorthand{:}`. (You can conveniently define your own macros, or even your own user shorthands provided they do not overlap.) For your records, here is a list of shorthands, but you must check them, as they may change:<sup>4</sup>

**Languages with no shorthands** Croatian, English (any variety), Indonesian, Hebrew, Interlingua, Irish, Lower Sorbian, Malaysian, North Sami, Romanian, Scottish, Welsh

**Languages with only " as defined shorthand character** Albanian, Bulgarian, Danish, Dutch, Finnish, German (old and new orthography, also Austrian), Icelandic, Italian, Norwegian, Polish, Portuguese (also Brazilian), Russian, Serbian (with Latin script), Slovene, Swedish, Ukrainian, Upper Sorbian

<sup>3</sup>Actually, any name not corresponding to a language group does the same as none. However, follow this convention because it might be enforced in future releases of babel to catch possible errors.

<sup>4</sup>Thanks to Enrico Gregorio

**Basque** " ' ~  
**Breton** : ; ? !  
**Catalan** " ' '  
**Czech** " -  
**Esperanto** ^  
**Estonian** " ~  
**French** (all varieties) : ; ? !  
**Galician** " . ' ~ < >  
**Greek** ~  
**Hungarian** '  
**Kurmanji** ^  
**Latin** " ^ =  
**Slovak** " ^ ' -  
**Spanish** " . < > '  
**Turkish** : ! =

In addition, the `babel` core declares `~` as a one-char shorthand which is let, like the standard `~`, to a non breaking space.<sup>5</sup>

## 1.4 Package options

**New 3.9** These package options are processed before language options, so that they are taken into account irrespective of its order. The first three options have been available in previous versions.

<b>KeepShorthandsActive</b>	Tells babel not to deactivate shorthands after loading a language file, so that they are also available in the preamble.
-----------------------------	--

**activeacute** For some languages babel supports this options to set ' as a shorthand in case it is not done by default.

activegrave Same for '.

shorthands=  $\langle char \rangle \langle char \rangle \dots$  | off

The only language shorthands activated are those given, like, eg:

```
\usepackage[esperanto,frenchb,shorthands=;!?]{babel}
```

If ' is included, activeacute is set; if ' is included, activegrave is set. Active characters (like ~) should be preceded by \string (otherwise they will be expanded by L<sup>A</sup>T<sub>E</sub>X before they are passed to the package and therefore they will not be recognized); however, t is provided for the common case of ~ (as well as c for not so common case of the comma).

With `shorthands=off` no language shorthands are defined, As some languages use this mechanism for tools not available otherwise, a macro `\babelshorthand` is defined, which allows using them; see above.

safe= none | ref | bib

Some L<sup>A</sup>T<sub>E</sub>X macros are redefined so that using shorthands is safe. With `safe=bib` only `\nocite`, `\bibcite` and `\bibitem` are redefined. With `safe=ref` only

---

<sup>5</sup>This declaration serves to nothing, but it is preserved for backward compatibility.

`\newlabel`, `\ref` and `\pageref` are redefined (as well as a few macros from `varioref` and `ifthen`). With `safe=none` no macro is redefined. Of course, in such a case you cannot use shorthands in these macros.

**math=** `active` | `normal`

Shorthands are mainly intended for text, not for math. By setting this option with the value `normal` they are deactivated in math mode (default is `active`) and things like `#{a'}` (a closing brace after a shorthand) are not a source of trouble any more.

**config=** `<file>`

Load `<file>.cfg` instead of the default config file `bblopts.cfg`.

**main=** `<language>`

Sets the main language, as explained above, ie, this language is always loaded last. If it is not given as package or global option, it is added to the list of requested languages.

**headfoot=** `<language>`

By default, headlines and footlines are not touched (only marks), and if they contain language dependent macros (which is not usual) there may be unexpected results. With this option you may set the language in heads and foots.

**noconfigs** Global and language default config files are not loaded, so you can make sure your document is not spoilt by an unexpected `.cfg` file. However, if the key `config` is set, this file is loaded.

**showlanguages** Prints to the log the list of languages loaded when the format was created: number (remember dialects can share it), name, hyphenation file and exceptions file.

**strings=** `generic` | `unicode` | `encoded` | `<label>` | `<font encoding>`

Selects the encoding of strings in languages supporting this feature. Predefined labels are `generic` (for traditional T<sub>E</sub>X, LICR and ASCII strings), `unicode` (for engines like xetex and luatex) and `encoded` (for special cases requiring mixed encodings). Other allowed values are font encoding codes (T1, T2A, LGR, L7X...), but only in languages supporting them.

**hyphenmap=** `off` | `main` | `select` | `other` | `other*`

**New 3.9g** Sets the behaviour of case mapping for hyphenation, provided the language defines it. It can take the following values:

**off** deactivates this feature and no case mapping is applied;

**first** sets it at the first switching commands in the current or parent scope (typically, when the aux file is first read and at `\begin{document}`}, but also the first `\selectlanguage` in the preamble), and it's the default if a single language option has been stated;<sup>6</sup>

**select** sets it only at `\selectlanguage`;

---

<sup>6</sup>Duplicated options count as several ones.

`other` also sets it at `otherlanguage`;  
`other*` also sets it at `otherlanguage*` as well as in heads and foots (if the option `headfoot` is used) and in auxiliary files (ie, at `\select@language`), and it's the default if several language options have been stated. The option `first` can be regarded as an optimized version of `other*` for monolingual documents.<sup>7</sup>

## 1.5 The base option

With this package option `babel` just loads some basic macros (those in `switch.def`), defines `\AfterBabelLanguage` and exits. It also selects the hyphenations patterns for the last language passed as option (by its name in `language.dat`). There are two main uses: classes and packages, and as a last resort in case there are, for some reason, incompatible languages. It can be used if you just want to select the hyphenations patterns of a single language, too.

`\AfterBabelLanguage`  $\{\langle option-name \rangle\}\{\langle code \rangle\}$

This command is currently the only provided by `base`. Executes  $\langle code \rangle$  when the file loaded by the corresponding package option is finished (at `\ldf@finish`). The setting is global. So

```
\AfterBabelLanguage{frenchb}{...}
```

does ... at the end of `frenchb.ldf`. It can be used in `ldf` files, too, but in such a case the code is executed only if  $\langle option-name \rangle$  is the same as `\CurrentOption` (which could not be the same as the option name as set in `\usepackage!`). For example, consider two languages `foo` and `bar` defining the same `\macro` with `\newcommand`. An error is raised if you attempt to load both. Here is a way to overcome this problem:

```
\usepackage[base]{babel}
\AfterBabelLanguage{foo}{%
  \let\macroFoo\macro
  \let\macro\relax}
\usepackage[foo,bar]{babel}
```

## 1.6 Hooks

**New 3.9** A hook is a piece of code to be executed at certain events. Some hooks are predefined when `luatex` and `xetex` are used.

`\AddBabelHook`  $\{\langle name \rangle\}\{\langle event \rangle\}\{\langle code \rangle\}$

The same name can be applied to several events. Hooks may be enabled and disabled for all defined events with `\EnableBabelHook{\langle name \rangle}`, `\DisableBabelHook{\langle name \rangle}`. Names containing the string `babel` are reserved (they are used, for example, by `\useshortands*` to add a hook for the event `afterextras`).

Current events are the following; in some of them you can use one or two  $\text{T}_{\text{E}}\text{X}$  parameters (`#1`, `#2`), with the meaning given:

<sup>7</sup>Providing foreign is pointless, because the case mapping applied is that at the end of paragraph, but if either `xetex` or `luatex` change this behaviour it might be added. On the other hand, `other` is provided even if I [JBL] think it isn't really useful, but who knows.

**addialect** (language name, dialect name) Used by `luababel.def` to load the patterns if not preloaded.

**patterns** (language name, language with encoding) Executed just after the `\language` has been set. The second argument has the patterns name actually selected (in the form of either `lang:ENC` or `lang`).

**hyphenation** (language name, language with encoding) Executed locally just before exceptions given in `\babelhyphenation` are actually set.

**defaultcommands** Used (locally) in `\StartBabelCommands`.

**encodedcommands** (input, font encodings) Used (locally) in `\StartBabelCommands`. Both xetex and luatex make sure the encoded text is read correctly.

**stopcommands** Used to reset the the above, if necessary.

**write** This event comes just after the switching commands are written to the aux file.

**beforeextras** Just before executing `\extras<language>`. This event and the next one should not contain language-dependent code (for that, add it to `\extras<language>`).

**afterextras** Just after executing `\extras<language>`. For example, the following deactivates shorthands in all languages:

```
\AddBabelHook{noshort}{\afterextras}{\languageshorthands{none}}
```

**stringprocess** Instead of a parameter, you can manipulate the macro `\BabelString` containing the string to be defined with `\SetString`. For example, to use an expanded version of the string in the definition, write:

```
\AddBabelHook{myhook}{stringprocess}{%
\protected@edef\BabelString{\BabelString}}
```

Four events are used in `hyphen.cfg`, which are handled in a quite different way for efficiency reasons – unlike the precedent ones, they only have a single hook and replace a default definition.

**everylanguage** (language) Executed before patterns are loaded.

**loadkernel** (file) By default loads `switch.def`. It can be used to load a different version of this files or to load nothing.

**loadpatterns** (patterns file) Loads the patterns file. Used by `luababel.def`.

**loadexceptions** (exceptions file) Loads the exceptions file. Used by `luababel.def`.

**\BabelContentsFiles** New 3.9 This macro contains a list of “toc” types which require a command to switch the language. Its default value is `toc,lof,lot`, but you may redefine it with `\renewcommand` (it’s up to you to make sure no toc type is duplicated).

## 1.7 Hyphenation tools

**\babelhyphen** \*`{<type>}`  
**\babelhyphen** \*`{<text>}`

New 3.9 It is customary to classify hyphens in two types: (1) *explicit* or *hard hyphens*, which in T<sub>E</sub>X are entered as `-`, and (2) *optional* or *soft hyphens*, which are entered as `\-`. Strictly, a *soft hyphen* is not a hyphen, but just a breaking opportunity or, in T<sub>E</sub>X terms, a “discretionary”; a *hard hyphen* is a hyphen with a

breaking opportunity after it. A further type is a *non-breaking hyphen*, a hyphen without a breaking opportunity.

In  $\TeX$ , - and \- forbid further breaking opportunities in the word. This is the desired behaviour very often, but not always, and therefore many languages provide shorthands for these cases. Unfortunately, this has not been done consistently: for example, in Dutch, Portugese, Catalan or Danish, " - is a hard hyphen, while in German, Spanish, Norwegian, Slovak or Russian, it is a soft hyphen. Furthermore, some of them even redefine \-, so that you cannot insert a soft hyphen without breaking opportunities in the rest of the word.

Therefore, some macros are provide with a set of basic “hyphens” which can be used by themselves, to define a user shorthand, or even in language files.

- `\babelhyphen{soft}` and `\babelhyphen{hard}` are self explanatory.
- `\babelhyphen{repeat}` inserts a hard hyphen which is repeated at the beginning of the next line, as done in languages like Polish, Portugese and Spanish.
- `\babelhyphen{nobreak}` inserts a hard hyphen without a break after it (even if a space follows).
- `\babelhyphen{empty}` inserts a break oportunity without a hyphen at all.
- `\babelhyphen{<text>}` is a hard “hyphen” using `<text>` instead. A typical case is `\babelhyphen{/}`.

With all of them hyphenation in the rest of the word is enabled. If you don’t want enabling it, there is a starred counterpart: `\babelhyphen*{soft}` (which in most cases is equivalent to the original \-), `\babelhyphen*{hard}`, etc.

Note hard is also good for isolated prefixes (eg, *anti-*) and nobreak for isolated suffixes (eg, *-ism*), but in both cases `\babelhyphen*{nobreak}` is usually better.

There are also some differences with  $\LaTeX$ : (1) the character used is that set for the current font, while in  $\LaTeX$  it is hardwired to - (a typical value); (2) the hyphen to be used in fonts with a negative `\hyphenchar` is -, like in  $\LaTeX$ , but it can be changed to another value by redefining `\babelnulhyphen`; (3) a break after the hyphen is forbidden if preceded by a glue  $>0$  pt (at the beginning of a word, provided it is not immediately preceded by, say, a parenthesis).

**`\babelhyphenation`** [`<language>`], [`<language>`], ... [`<exceptions>`]

**New 3.9** Sets hyphenation exceptions for the languages given or, without the optional argument, for *all* languages (eg, proper nouns or common loan words, and of course monolingual documents). Language exceptions take precedence over global ones.

It can be used only in the preamble, and exceptions are set when the language is first selected, thus taking into account changes of `\lccodes`’s done in `\extras<lang>` as well as the script specific encoding (not set in the preamble by default). Multiple `\babelhyphenation`’s are allowed. For example:

```
\babelhyphenation{Wal-hal-la Dar-bhan-ga}
```

Listed words are saved expanded and therefore it relies on the LICR. Of course, it also works without the LICR if the input and the font encondings are the same, like in Unicode based engines.

## 1.8 Language attributes

**\languageattribute** This is a user-level command, to be used in the preamble of a document (after `\usepackage[...]{babel}`), that declares which attributes are to be used for a given language. It takes two arguments: the first is the name of the language; the second, a (list of) attribute(s) to be used. Attributes must be set in the preamble and only once – they cannot be turned on and off. The command checks whether the language is known in this document and whether the attribute(s) are known for this language.

Very often, using a *modifier* in a package option is better.

Several language definition files use their own methods to set options. For example, frenchb uses `\frenchbsetup`, magyar (1.5) uses `\magyarOptions`; modifiers provided by spanish have no attribute counterparts. Macros setting options are also used (eg, `\ProsodicMarksOn` in latin).

## 1.9 Languages supported by babel

In the following table most of the languages supported by babel are listed, together with the names of the options which you can load babel with for each language. Note this list is open and the current options may be different.

<b>Afrikaans</b>	afrikaans
<b>Bahasa</b>	bahasa, indonesian, indon, bahasai, bahasam, malay, meyalu
<b>Basque</b>	basque
<b>Breton</b>	breton
<b>Bulgarian</b>	bulgarian
<b>Catalan</b>	catalan
<b>Croatian</b>	croatian
<b>Czech</b>	czech
<b>Danish</b>	danish
<b>Dutch</b>	dutch
<b>English</b>	english, USenglish, american, UKenglish, british, canadian, australian, newzealand
<b>Esperanto</b>	esperanto
<b>Estonian</b>	estonian
<b>Finnish</b>	finnish
<b>French</b>	french, francais, canadien, acadian
<b>Galician</b>	galician
<b>German</b>	austrian, german, germanb, ngerman, naustrian
<b>Greek</b>	greek, polutonikogreek
<b>Hebrew</b>	hebrew
<b>Icelandic</b>	icelandic
<b>Interlingua</b>	interlingua
<b>Irish Gaelic</b>	irish
<b>Italian</b>	italian
<b>Latin</b>	latin
<b>Lower Sorbian</b>	lowersorbian
<b>North Sami</b>	samin
<b>Norwegian</b>	norsk, nynorsk
<b>Polish</b>	polish
<b>Portuguese</b>	portuges, portuguese, brazilian, brazil
<b>Romanian</b>	romanian

**Russian** russian  
**Scottish Gaelic** scottish  
**Spanish** spanish  
**Slovakian** slovak  
**Slovenian** slovene  
**Swedish** swedish  
**Serbian** serbian  
**Turkish** turkish  
**Ukrainian** ukrainian  
**Upper Sorbian** uppersorbian  
**Welsh** welsh

There are more languages not listed above, including hindi, thai, thaicjk, latvian, turkmen, magyar, mongolian, romansh, lithuanian, spanglish, vietnamese, japanese, pinyin, arabic, farsi, ibygreek, bgreek, serbianc, frenchle, ethiop and friulan.

Most of them work out of the box, but some may require extra fonts, encoding files, a preprocessor or even a complete framework (like CJK). For example, if you have got the velthuis/devnag package, you can create a file with extension .dn:

```

\documentclass{article}
\usepackage[hindi]{babel}
\begin{document}
{\dn devaanaa.m priya.h}
\end{document}

```

Then you preprocess it with devnag  $\langle file \rangle$ , which creates  $\langle file \rangle.tex$ ; you can then typeset the latter with L<sup>A</sup>T<sub>E</sub>X.

## 1.10 Tips, workarounds, know issues and notes

- If you use the document class book *and* you use `\ref` inside the argument of `\chapter` (or just use `\ref` inside `\MakeUppercase`), L<sup>A</sup>T<sub>E</sub>X will keep complaining about an undefined label. To prevent such problems, you could revert to using uppercase labels, you can use `\lowercase{\ref{foo}}` inside the argument of `\chapter`, or, if you will not use shorthands in labels, set the safe option to none or bib.
- Both ltxdoc and babel use `\AtBeginDocument` to change some catcodes, and babel reloads hline to make sure : has the right one, so if you want to change the catcode of | it has to be done using the same method at the proper place, with

```
\AtBeginDocument{\DeleteShortVerb{\|}}
```

*before* loading babel. This way, when the document begins the sequence is (1) make | active (ltxdoc); (2) make it unactive (your settings); (3) make babel shorthands active (babel); (4) reload hline (babel, now with the correct catcodes for | and :).

- Documents with several input encodings are not frequent, but sometimes are useful. You can set different encodings for different languages as the following example shows:



```
\addto\extrasfrench{\inputencoding{latin1}}
\addto\extrasrussian{\inputencoding{koi8-r}}
```

(A recent version of inputenc is required.)

- For the hyphenation to work correctly, lccodes cannot change, because  $\TeX$  only takes into account the values when the paragraph is hyphenated, i.e., when it has been finished.<sup>8</sup> So, if you write a chunk of French text with `\foreignlanguage`, the apostrophes might not be taken into account. This is a limitation of  $\TeX$ , not of babel. Alternatively, you may use `\usesorthands` to activate `'` and `\defineshorthand`, or redefine `\textquoteright` (the latter is called by the non-ASCII right quote).
- `\bibitem` is out of sync with `\selectlanguage` in the .aux file. The reason is `\bibitem` uses `\immediate` (and others, in fact), while `\selectlanguage` doesn't. There is no known workaround.
- Babel does not take into account `\normalsfcodes` and (non-)French spacing is not always properly (un)set by languages. However, problems are unlikely to happen and therefore this part remains untouched in version 3.9 (but it is in the 'to do' list).
- Using a character mathematically active (ie, with math code "8000) as a shorthand can make  $\TeX$  enter in an infinite loop. (Another issue in the 'to do' list, although there is a partial solution.)
- Also in the 'to do' list is a common interface to switch scripts, to avoid the current problem of languages trying to define `\text<script>` in different ways.

The following packages can be useful, too (the list is still far from complete):

**csquotes** Logical markup for quotes.

**iflang** Tests correctly the current language.

**hyphsubst** Selects a different set of patterns for a language.

**translator** An open platform for packages that need to be localized.

**siunitx** Typesetting of numbers and physical quantities.

**biblatex** Programmable bibliographies and citations.

**bicaption** Bilingual captions.

**babelbib** Multilingual bibliographies

## 1.11 Future work

Useful additions would be, for example, time, currency, addresses and personal names.<sup>9</sup> But that is the easy part, because they don't require modifying the  $\LaTeX$  internals.

More interesting are differences in the sentence structure or related to it. For example, in Basque the number precedes the name (including chapters), in Hungarian "from (1)" is "(1)-ból", but "from (3)" is "(3)-ból", in Spanish an item labelled "3." may be referred to as either "ítem 3.<sup>o</sup>" or "3.<sup>er</sup> ítem", and so on.

<sup>8</sup>This explains why  $\LaTeX$  assumes the lowercase mapping of T1 and does not provide a tool for multiple mappings. Unfortunately, `\savingshyphcodes` is not a solution either, because lccodes for hyphenation are frozen in the format and cannot be changed.

<sup>9</sup>See for example POSIX, ISO 14652 and the Unicode Common Locale Data Repository (CLDR).

Even more interesting is right-to-left, vertical and bidi typesetting. Babel provided a basic support for bidi text as part of the style for Hebrew, but it is somewhat unsatisfactory and internally replaces some hardwired commands by other hardwired commands (generic changes would be much better).

Handling of Unicode fonts is also problematic. There is `fontspec`, but special macros are required (not only the NFSS ones) and it doesn't provide "orthogonal axis" for features, including those related to the language (mainly language and script). A couple of tentative macros, which solve the two main cases, are provided by babel with a partial solution (only xetex and luatex, for obvious reasons), but use them at your own risk, as they might be removed in the future. For this very reason, they are described here:

- `\babelFSstore{<babel-language>}` sets the current three basic families (rm, sf, tt) as the default for the language given. In most cases, this macro will be enough.
- `\babelFSdefault{<babel-language>}{<fontspec-features>}` patches `\fontspec` so that the given features are always passed as the optional argument or added to it (not an ideal solution). Use it only if you select some fonts in the document with `\fontspec`.

So, for example:

```
\setmainfont[Language=Turkish]{Minion Pro}
\setsansfont[Language=Turkish]{Myriad Pro}
\babelFSstore{turkish}
\setmainfont{Minion Pro}
\setsansfont{Myriad Pro}
\babelFSfeatures{turkish}{Language=Turkish}
```

Note you can set any feature required for the language – not only Language, but also Script or a local .fea. This makes those macros a bit more verbose, but also more powerful.

## 2 Preloading languages with language.dat

T<sub>E</sub>X and most engines based on it (pdfT<sub>E</sub>X, xetex,  $\epsilon$ -T<sub>E</sub>X, the main exception being luatex) require hyphenation patterns to be loaded when a format is created (eg, L<sup>A</sup>T<sub>E</sub>X, XeL<sup>A</sup>T<sub>E</sub>X, pdfL<sup>A</sup>T<sub>E</sub>X). babel provides a tool which has become standard in many distributions and based on a "configuration file" named `language.dat`. The exact way this file is used depends on the distribution, so please, read the documentation for the latter (note also some distributions generate the file with some tool).

In that file the person who maintains a T<sub>E</sub>X environment has to record for which languages he has hyphenation patterns *and* in which files these are stored<sup>10</sup>. When hyphenation exceptions are stored in a separate file this can be indicated by naming that file *after* the file with the hyphenation patterns.

The file can contain empty lines and comments, as well as lines which start with an equals (=) sign. Such a line will instruct L<sup>A</sup>T<sub>E</sub>X that the hyphenation patterns just processed have to be known under an alternative name. Here is an example:

```
% File : language.dat
```

<sup>10</sup>This is because different operating systems sometimes use very different file-naming conventions.

```
% Purpose : tell iniTeX what files with patterns to load.
english    english.hyphenations
=british

dutch      hyphen.dutch exceptions.dutch % Nederlands
german hyphen.ger
```

You may also set the font encoding the patterns are intended for by following the language name by a colon and the encoding code.<sup>11</sup> For example:

```
german:T1 hyphenT1.ger
german hyphen.ger
```

With the previous settings, if the encoding when the language is selected is T1 then the patterns in hyphenT1.ger are used, but otherwise use those in hyphen.ger (note the encoding could be set in `\extras{lang}`).

A typical error when using babel is the following:

```
No hyphenation patterns were preloaded for
the language '<lang>' into the format.
Please, configure your TeX system to add them and
rebuild the format. Now I will use the patterns
preloaded for english instead}}
```

It simply means you must reconfigure language.dat, either by hand or with the tools provided by your distribution.

### 3 The interface between the core of babel and the language definition files

The *language definition files* (ldf) must conform to a number of conventions, because these files have to fill in the gaps left by the common code in babel.def, i. e., the definitions of the macros that produce texts. Also the language-switching possibility which has been built into the babel system has its implications.

The following assumptions are made:

- Some of the language-specific definitions might be used by plain T<sub>E</sub>X users, so the files have to be coded so that they can be read by both L<sup>A</sup>T<sub>E</sub>X and plain T<sub>E</sub>X. The current format can be checked by looking at the value of the macro `\fmtname`.
- The common part of the babel system redefines a number of macros and environments (defined previously in the document style) to put in the names of macros that replace the previously hard-wired texts. These macros have to be defined in the language definition files.
- The language definition files must define five macros, used to activate and deactivate the language-specific definitions. These macros are `\<lang>hyphenmins`, `\captions<lang>`, `\date<lang>`, `\extras<lang>` and `\noextras<lang>` (the last two may be left empty); where `<lang>` is either the name of the language definition file or the name of the L<sup>A</sup>T<sub>E</sub>X option that is to be used. These macros and their functions are discussed below. You must define all or none for a language (or a dialect); defining, say, `\date<lang>` but

<sup>11</sup>This is not a new feature, but in former versions it didn't work correctly.

not `\captions⟨lang⟩` does not raise an error but can lead to unexpected results.

- When a language definition file is loaded, it can define `\l@⟨lang⟩` to be a dialect of `\language0` when `\l@⟨lang⟩` is undefined.
- Language names must be all lowercase. If an unknown language is selected, babel will attempt setting it after lowercasing its name.
- The semantics of modifiers is not defined (on purpose). In most cases, they will just be simple separated options (eg., `spanish`), but a language might require, say, a set of options organized as a tree with suboptions (in such a case, the recommended separator is `/`).

Some recommendations:

- The preferred shorthand is `"`, which is not used in L<sup>A</sup>T<sub>E</sub>X (quotes are entered as `'` and `'`). Other good choices are characters which are not used in a certain context (eg, `=` in an ancient language). Note however `=`, `<`, `>`, `:` and the like can be dangerous, because they may be used as part of the syntax of some elements (numeric expressions, key/value pairs, etc.).
- Captions should not contain shorthands or encoding dependent commands (the latter is not always possible, but should be clearly documented). They should be defined using the LICR. You may also use the new tools for encoded strings, described below.
- Avoid adding things to `\noextras⟨lang⟩` except for `umlauthigh` and friends, `\bbl@deactivate`, `\bbl@(non)frenchspacing`, and language specific macros. Use always, if possible, `\bbl@save` and `\bbl@savevariable` (except if you still want to have access to the previous value).

There are no special requirements for documenting your language files. Now they are not included in the base babel manual, so provide a standalone document suited for your needs (and the corresponding PDF, if you like), as well as other files you think can be useful (eg, `samples`, `readme`).

### 3.1 Basic macros

In the core of the babel system, several macros are defined for use in language definition files. Their purpose is to make a new language known. The first two are related to hyphenation patterns.

**`\addlanguage`** The macro `\addlanguage` is a non-outer version of the macro `\newlanguage`, defined in `plain.tex` version 3.x. For older versions of `plain.tex` and `lplain.tex` a substitute definition is used. Here “language” is used in the T<sub>E</sub>X sense of set of hyphenation patterns.

**`\adddialect`** The macro `\adddialect` can be used when two languages can (or must) use the same hyphenation patterns. This can also be useful for languages for which no patterns are preloaded in the format. In such cases the default behaviour of the babel system is to define this language as a ‘dialect’ of the language for which the patterns were loaded as `\language0`. Here “language” is used in the T<sub>E</sub>X sense of set of hyphenation patterns.

**`\<lang>hyphenmins`** The macro `\⟨lang⟩hyphenmins` is used to store the values of the `\lefthyphenmin` and `\righthyphenmin`. Redefine this macro to set your own values, with two numbers corresponding to these two parameters. For example:

```
\renewcommand\spanishhyphenmins{34}
```

(Assigning `\lefthyphenmin` and `\righthyphenmin` directly in `\extras<lang>` has no effect.)

`\providehyphenmins` The macro `\providehyphenmins` should be used in the language definition files to set `\lefthyphenmin` and `\righthyphenmin`. This macro will check whether these parameters were provided by the hyphenation file before it takes any action. If these values have been already set, this command is ignored (currently, default pattern files do *not* set them).

`\captions<lang>` The macro `\captions<lang>` defines the macros that hold the texts to replace the original hard-wired texts.

`\date<lang>` The macro `\date<lang>` defines `\today`.

`\extras<lang>` The macro `\extras<lang>` contains all the extra definitions needed for a specific language. This macro, like the following, is a hook – you can add things to it, but it must not be used directly.

`\noextras<lang>` Because we want to let the user switch between languages, but we do not know what state  $\TeX$  might be in after the execution of `\extras<lang>`, a macro that brings  $\TeX$  into a predefined state is needed. It will be no surprise that the name of this macro is `\noextras<lang>`.

`\bbl@declare@ttribute` This is a command to be used in the language definition files for declaring a language attribute. It takes three arguments: the name of the language, the attribute to be defined, and the code to be executed when the attribute is to be used.

`\main@language` To postpone the activation of the definitions needed for a language until the beginning of a document, all language definition files should use `\main@language` instead of `\selectlanguage`. This will just store the name of the language, and the proper language will be activated at the start of the document.

`\ProvidesLanguage` The macro `\ProvidesLanguage` should be used to identify the language definition files. Its syntax is similar to the syntax of the  $\LaTeX$  command `\ProvidesPackage`.

`\LdfInit` The macro `\LdfInit` performs a couple of standard checks that must be made at the beginning of a language definition file, such as checking the category code of the `@`-sign, preventing the `.ldf` file from being processed twice, etc.

`\ldf@quit` The macro `\ldf@quit` does work needed if a `.ldf` file was processed earlier. This includes resetting the category code of the `@`-sign, preparing the language to be activated at `\begin{document}` time, and ending the input stream.

`\ldf@finish` The macro `\ldf@finish` does work needed at the end of each `.ldf` file. This includes resetting the category code of the `@`-sign, loading a local configuration file, and preparing the language to be activated at `\begin{document}` time.

`\loadlocalcfg` After processing a language definition file,  $\LaTeX$  can be instructed to load a local configuration file. This file can, for instance, be used to add strings to `\captions<lang>` to support local document classes. The user will be informed that this configuration file has been loaded. This macro is called by `\ldf@finish`.

`\substitutefontfamily` (Deprecated.) This command takes three arguments, a font encoding and two font family names. It creates a font description file for the first font in the given encoding. This `.fd` file will instruct  $\LaTeX$  to use a font from the second family when a font from the first family in the given encoding seems to be needed.

## 3.2 Skeleton

Here is the basic structure of an `ldf` file, with a language, a dialect and an attribute. Strings are best defined using the method explained in in sec. 3.7 (babel

3.9 and later).

```
\ProvidesLanguage{<language>}
  [0000/00/00 v0.0 <Language> support from the babel system]
\LdfInit{<language>}{captions<language>}

\ifx\undefined\l@<language>
  \nopatterns{<Language>}
  \adddialect\l@<language>0
\fi

\adddialect\l@<dialect>\l@<language>

\babel@declare@attribute{<language>}{<attrib>}{%
  \expandafter\addto\expandafter\extras<language>
  \expandafter{\extras<attrib><language>}%
  \let\captions<language>\captions<attrib><language>}

\providehyphenmins{<language>}{\tw@\thr@@}

\StartBabelCommands*{<language>}{captions}
\SetString\chaptername{<chapter name>}
% More strings

\StartBabelCommands*{<language>}{date}
\SetString\monthinname{<name of first month>}
% More strings

\StartBabelCommands*{<dialect>}{captions}
\SetString\chaptername{<chapter name>}
% More strings

\StartBabelCommands*{<dialect>}{date}
\SetString\monthinname{<name of first month>}
% More strings

\EndBabelCommands

\addto\extras<language>{}
\addto\noextras<language>{}
\let\extras<dialect>\extras<language>
\let\noextras<dialect>\noextras<language>

\ldf@finish{<language>}
```

### 3.3 Support for active characters

In quite a number of language definition files, active characters are introduced. To facilitate this, some support macros are provided.

`\initiate@active@char`

The internal macro `\initiate@active@char` is used in language definition files to instruct L<sup>A</sup>T<sub>E</sub>X to give a character the category code ‘active’. When a character has been made active it will remain that way until the end of the document. Its definition may vary.

`\bbl@activate`  
`\bbl@deactivate`

The command `\bbl@activate` is used to change the way an active character expands. `\bbl@activate` ‘switches on’ the active behaviour of the character.

`\bbl@deactivate` lets the active character expand to its former (mostly) non-active self.

`\declare@shorthand` The macro `\declare@shorthand` is used to define the various shorthands. It takes three arguments: the name for the collection of shorthands this definition belongs to; the character (sequence) that makes up the shorthand, i.e. `~` or `"a`; and the code to be executed when the shorthand is encountered. (It does *not* raise an error if the shorthand character has not been “initiated”.)

`\bbl@add@special`  
`\bbl@remove@special` The  $\TeX$ book states: “Plain  $\TeX$  includes a macro called `\dospecials` that is essentially a set macro, representing the set of all characters that have a special category code.” [1, p. 380] It is used to set text ‘verbatim’. To make this work if more characters get a special category code, you have to add this character to the macro `\dospecials`.  $\LaTeX$  adds another macro called `\@sanitize` representing the same character set, but without the curly braces. The macros `\bbl@add@special<char>` and `\bbl@remove@special<char>` add and remove the character `<char>` to these two sets.

### 3.4 Support for saving macro definitions

Language definition files may want to redefine macros that already exist. Therefore a mechanism for saving (and restoring) the original definition of those macros is provided. We provide two macros for this<sup>12</sup>.

`\babel@save` To save the current meaning of any control sequence, the macro `\babel@save` is provided. It takes one argument, `<csname>`, the control sequence for which the meaning has to be saved.

`\babel@savevariable` A second macro is provided to save the current value of a variable. In this context, anything that is allowed after the `\the` primitive is considered to be a variable. The macro takes one argument, the `<variable>`.  
The effect of the preceding macros is to append a piece of code to the current definition of `\originalTeX`. When `\originalTeX` is expanded, this code restores the previous definition of the control sequence or the previous value of the variable.

### 3.5 Support for extending macros

`\addto` The macro `\addto{<control sequence>}{< $\TeX$  code>}` can be used to extend the definition of a macro. The macro need not be defined (ie, it can be undefined or `\relax`). This macro can, for instance, be used in adding instructions to a macro like `\extrasenglish`.

Be careful when using this macro, because depending on the case the assignment could be either global (usually) or local (sometimes). That does not seem very consistent, but this behaviour is preserved for backward compatibility. If you are using `etoolbox`, by Philipp Lehman, consider using the tools provided by this package instead of `\addto`.

### 3.6 Macros common to a number of languages

`\bbl@allowhyphens` In several languages compound words are used. This means that when  $\TeX$  has to hyphenate such a compound word, it only does so at the ‘-’ that is used in such words. To allow hyphenation in the rest of such a compound word, the macro `\bbl@allowhyphens` can be used.

<sup>12</sup>This mechanism was introduced by Bernd Raichle.



<code>\allowhyphens</code>	Same as <code>\bbl@allowhyphens</code> , but does nothing if the encoding is T1. It is intended mainly for characters provided as real glyphs by this encoding but constructed with <code>\accent</code> in OT1. Note the previous command ( <code>\bbl@allowhyphens</code> ) has different applications (hyphens and discretionaries) than this one (composite chars). Note also prior to version 3.7, <code>\allowhyphens</code> had the behaviour of <code>\bbl@allowhyphens</code> .
<code>\set@low@box</code>	For some languages, quotes need to be lowered to the baseline. For this purpose the macro <code>\set@low@box</code> is available. It takes one argument and puts that argument in an <code>\hbox</code> , at the baseline. The result is available in <code>\box0</code> for further processing.
<code>\save@sf@q</code>	Sometimes it is necessary to preserve the <code>\spacefactor</code> . For this purpose the macro <code>\save@sf@q</code> is available. It takes one argument, saves the current <code>\spacefactor</code> , executes the argument, and restores the <code>\spacefactor</code> .
<code>\bbl@frenchspacing</code> <code>\bbl@nonfrenchspacing</code>	The commands <code>\bbl@frenchspacing</code> and <code>\bbl@nonfrenchspacing</code> can be used to properly switch French spacing on and off.

### 3.7 Encoding-dependent strings

**New 3.9** Babel 3.9 provides a way of defining strings in several encodings, intended mainly for `luatex` and `xetex`. This is the only new feature requiring changes in language files if you want to make use of it.

Furthermore, it must be activated explicitly, with the package option `strings`. If there is no `strings`, these blocks are ignored, except `\SetCases` (and except if forced with `\UseStrings`, see below). In other words, the old way of defining/switching strings still works and it's used by default.

It consist is a series of blocks started with `\StartBabelCommands`. The last block is closed with `\EndBabelCommands`. Each block is a single group (ie, local declarations apply until the next `\StartBabelCommands` or `\EndBabelCommands`). An `ldf` may contain several series of this kind.

Thanks to this new feature, string values and string language switching are not mixed any more. No need of `\addto`. If the language is german, just redefine `\germanchaptername`.

`\StartBabelCommands`  $\{ \langle \textit{language-list} \rangle \} \{ \langle \textit{category} \rangle \} [ \langle \textit{selector} \rangle ]$

The  $\langle \textit{language-list} \rangle$  specifies which languages the block is intended for. A block is taken into account only if the `\CurrentOption` is listed here. Alternatively, you can define `\BabelLanguages` to a comma-separated list of languages to be defined (if undefined, `\StartBabelCommands` sets it to `\CurrentOption`). You may write `\CurrentOption` as the language, but this discouraged – a explicit name (or names) is much better and clearer.

A “selector” is a name to be used as value in package option `strings`, optionally followed by extra info about the encodings to be used. The name `unicode` must be used for `xetex` and `luatex` (the key `strings` has also other two special values: `generic` and `encoded`).

If a string is set several times (because several blocks are read), the first one take precedence (ie, it works much like `\providecommand`).

Encoding info is `charset=` followed by a `charset`, which if given sets how the strings should be traslated to the internal representation used by the engine, typically `utf8`, which is the only value supported currently (default is no traslations). Note `charset` is applied by `luatex` and `xetex` when reading the file, not when the macro or string is used in the document.



A list of font encodings which the strings are expected to work with can be given after `fontenc=` (separated with spaces, if two or more) – recommended, but not mandatory, although blocks without this key are not taken into account if you have requested `strings=encoded`.

Blocks without a selector are read always if the key `strings` has been used. They provide fallback values, and therefore must be the last blocks; they should be provided always if possible and all strings should be defined somehow inside it; they can be the only blocks (mainly LGC scripts using the LICR). Blocks without a selector can be activated explicitly with `strings=generic` (no block is taken into account except those). With `strings=encoded`, strings in those blocks are set as default (internally, `?`). If there is no key `strings`, string definitions are ignored, but `\SetCases` are still honoured (an a encoded way).

The `<category>` is either captions, date or extras. You must stick to these three categories, even if no error is raised when using other name.<sup>13</sup> It may be empty, too, but in such a case using `\SetString` is an error (but not `\SetCase`).

```
\StartBabelCommands{language}{captions}
  [unicode, fontenc=EU1 EU2, charset=utf8]
\SetString{\chaptername}{utf8-string}

\StartBabelCommands{language}{captions}
\SetString{\chaptername}{ascii-maybe-LICR-string}

\EndBabelCommands
```

A real example is:

```
\StartBabelCommands{austrian}{date}
  [unicode, fontenc=EU1 EU2, charset=utf8]
\SetString\monthinname{Jänner}

\StartBabelCommands{german,austrian}{date}
  [unicode, fontenc=EU1 EU2, charset=utf8]
\SetString\monthiiname{März}

\StartBabelCommands{austrian}{date}
\SetString\monthinname{J\"a}nner}

\StartBabelCommands{german}{date}
\SetString\monthinname{Januar}

\StartBabelCommands{german,austrian}{date}
\SetString\monthiiname{Februar}
\SetString\monthiiname{M\"a}rz}
\SetString\monthivname{April}
\SetString\monthvname{Mai}
\SetString\monthviname{Juni}
\SetString\monthviiname{Juli}
\SetString\monthviiiname{August}
\SetString\monthixname{September}
\SetString\monthxname{Oktober}
\SetString\monthxiname{November}
\SetString\monthxiiname{Dezenber}
\SetString\today{\number\day.~%
```

<sup>13</sup>In future releases further categories may be added.

```
\csname month\romannumeral\month name\endcsname\space
\number\year}
```

```
\StartBabelCommands{german,austrian}{captions}
  \SetString\prefacename{Vorwort}
  [etc.]
```

```
\EndBabelCommands
```

When used in ldf files, previous values of `\langle category \rangle \langle language \rangle` are overridden, which means the old way to define strings still works and used by default (to be precise, is first set to undefined and then strings are added). However, when used in the preamble or in a package, new settings are added to the previous ones, if the language exists (in the babel sense, ie, if `\date \langle language \rangle` exists).

**\StartBabelCommands** `*{\langle language-list \rangle}{\langle category \rangle}[\langle selector \rangle]`

The starred version just forces strings to take a value – if not set as package option, then the default for the engine is used. This is not done by default to prevent backward incompatibilities, but if you are creating a new language this version is better. It’s up to the maintainers of the current languages to decide if using it is appropriate.<sup>14</sup>

**\EndBabelCommands** Marks the end of the series of blocks.

**\AfterBabelCommands** `{\langle code \rangle}`

The code is delayed and executed at the global scope just after `\EndBabelCommands`.

**\SetString** `{\langle macro-name \rangle}{\langle string \rangle}`

Adds `\langle macro-name \rangle` to the current category, and defines globally `\langle lang-macro-name \rangle` to `\langle code \rangle` (after applying the transformation corresponding to the current charset or defined with the hook `stringprocess`). Use this command to define strings, without including any “logic” if possible, which should be a separated macro. See the example above for the date.

**\SetStringLoop** `{\langle macro-name \rangle}{\langle string-list \rangle}`

A convenient way to define several ordered names at once. For example, to define `\abmoniname`, `\abmoniiname`, etc. (and similarly with `abday`):

```
\SetStringLoop{abmon#1name}{en,fb,mr,ab,my,jn,jl,ag,sp,oc,nv,dc}
\SetStringLoop{abday#1name}{lu,ma,mi,ju,vi,sa,do}
```

`#1` is replaced by the roman numeral.

**\SetCase** `[\langle map-list \rangle]{\langle toupper-code \rangle}{\langle tolower-code \rangle}`

Sets globally code to be executed at `\MakeUppercase` and `\MakeLowercase`. The code would be typically things like `\let\BB\bb` and `\uccode` or `\lccode` (although for the reasons explained above, changes in lc/uc codes may not work). A `\langle map-list \rangle` is a series of macros using the internal format of `@uclclist` (eg,

<sup>14</sup>This replaces in 3.9g a short-lived `\UseStrings` which has been removed because it did not work.

`\bb\BB\cc\CC`). The mandatory arguments take precedence over the optional one. This command, unlike `\SetString`, is executed always (even without strings), and it is intended for minor readjustments only.

For example, as T1 is the default case mapping in L<sup>A</sup>T<sub>E</sub>X, we could set for Turkish:

```
\StartBabelCommands{turkish}{}[ot1enc, fontenc=OT1]
\SetCase
  {\uccode"10='I\relax}
  {\lccode'I="10\relax}

\StartBabelCommands{turkish}{}[unicode, fontenc=EU1 EU2, charset=utf8]
\SetCase
  {\uccode'i='İ\relax
   \uccode'ı='I\relax}
  {\lccode'İ='i\relax
   \lccode'I='ı\relax}

\StartBabelCommands{turkish}{}
\SetCase
  {\uccode'i="9D\relax
   \uccode"19='I\relax}
  {\lccode"9D='i\relax
   \lccode'I="19\relax}

\EndBabelCommands
```

(Note the mapping for OT1 is not complete.)

`\SetHyphenMap` `{\to-lower-macros}`

**New 3.9g** Case mapping serves in T<sub>E</sub>X for two unrelated purposes: case transforms (upper/lower) and hyphenation. `\SetCase` handles the former, while hyphenation is handled by `\SetHyphenMap` and controlled with the package option `hyphenmap`. So, even if internally they are based on the same T<sub>E</sub>X primitive (`\lccode`), babel sets them separately.

There are three helper macros to be used inside `\SetHyphenMap`:

- `\BabelLower{\uccode}{\lccode}` is similar to `\lccode` but it's ignored if the char has been set and saves the original `\lccode` to restore it when switching the language (except with `hyphenmap=first`).
- `\BabelLowerMM{\uccode-from}{\uccode-to}{\step}{\lccode-from}` loops though the given uppercase codes, using the step, and assigns them the `\lccode`, which is also increased (MM stands for *many-to-many*).
- `\BabelLowerM0{\uccode-from}{\uccode-to}{\step}{\lccode}` loops though the given uppercase codes, using the step, and assigns them the `\lccode`, which is fixed (M0 stands for *many-to-one*).

An example is (which is redundant, because these assignments are done by both `luatex` and `xetex`):

```
\SetHyphenMap{\BabelLowerMM{"100}{ "11F}{2}{ "101}}
```

This macro is not intended to fix wrong mappings done by Unicode (which are the default in both `xetex` and `luatex`) – if an assignment is wrong, fix it directly.

## 4 Compatibility and changes

### 4.1 Compatibility with `german.sty`

The file `german.sty` has been one of the sources of inspiration for the `babel` system. Because of this I wanted to include `german.sty` in the `babel` system. To be able to do that I had to allow for one incompatibility: in the definition of the macro `\selectlanguage` in `german.sty` the argument is used as the  $\langle number \rangle$  for an `\ifcase`. So in this case a call to `\selectlanguage` might look like `\selectlanguage{\german}`.

In the definition of the macro `\selectlanguage` in `babel.def` the argument is used as a part of other macronames, so a call to `\selectlanguage` now looks like `\selectlanguage{german}`. Notice the absence of the escape character. As of version 3.1a of `babel` both syntaxes are allowed.

All other features of the original `german.sty` have been copied into a new file, called `germanb.sty`<sup>15</sup>.

Although the `babel` system was developed to be used with L<sup>A</sup>T<sub>E</sub>X, some of the features implemented in the language definition files might be needed by plain T<sub>E</sub>X users. Care has been taken that all files in the system can be processed by plain T<sub>E</sub>X.

### 4.2 Compatibility with `ngerman.sty`

When used with the options `ngerman` or `naustrian`, `babel` will provide all features of the package `ngerman`. There is however one exception: The commands for special hyphenation of double consonants ("ff etc.) and ck ("ck), which are no longer required with the new German orthography, are undefined. With the `ngerman` package, however, these commands will generate appropriate warning messages only.

### 4.3 Compatibility with the french package

It has been reported to me that the package `french` by Bernard Gaulle ([gaulle@idris.fr](mailto:gaulle@idris.fr)) works together with `babel`. On the other hand, it seems *not* to work well together with a lot of other packages. Therefore I have decided to no longer load `french.lfd` by default. Instead, when you want to use the package by Bernard Gaulle, you will have to request it specifically, by passing either `frenchle` or `frenchpro` as an option to `babel`.

### 4.4 Changes in `babel` version 3.9

Most of changes in version 3.9 are related to bugs, either to fix them (there were lots), or to provide some alternatives. Even new features like `\babelhyphen` are intended to solve a certain problem (in this case, the lacking of a uniform syntax and behaviour for shorthands across languages). These changes are described in this manual in the correspondin place.

### 4.5 Changes in `babel` version 3.7

In `babel` version 3.7 a number of bugs that were found in version 3.6 are fixed. Also a number of changes and additions have occurred:

---

<sup>15</sup>The 'b' is added to the name to distinguish the file from Partls' file.

- Shorthands are expandable again. The disadvantage is that one has to type `'{ }a` when the acute accent is used as a shorthand character. The advantage is that a number of other problems (such as the breaking of ligatures, etc.) have vanished.
- Two new commands, `\shorthandon` and `\shorthandoff` have been introduced to enable to temporarily switch off one or more shorthands.
- Support for typesetting Greek has been enhanced. Code from the `kdgree` package (suggested by the author) was added and `\greeknumeral` has been added.
- Support for typesetting Basque is now available thanks to Juan Aguirregabiria.
- Support for typesetting Serbian with Latin script is now available thanks to Dejan Muhamedagić and Jankovic Slobodan.
- Support for typesetting Hebrew (and potential support for typesetting other right-to-left written languages) is now available thanks to Rama Porrat and Boris Lavva.
- Support for typesetting Bulgarian is now available thanks to Georgi Boshnakov.
- Support for typesetting Latin is now available, thanks to Claudio Beccari and Krzysztof Konrad Żelechowski.
- Support for typesetting North Sami is now available, thanks to Regnor Jernsletten.
- The options `canadian`, `canadien` and `acadien` have been added for Canadian English and French use.
- A language attribute has been added to the `\mark . . .` commands in order to make sure that a Greek header line comes out right on the last page before a language switch.
- Hyphenation pattern files are now read *inside a group*; therefore any changes a pattern file needs to make to lowercase codes, uppercase codes, and category codes are kept local to that group. If they are needed for the language, these changes will need to be repeated and stored in `\extras . . .`
- The concept of language attributes is introduced. It is intended to give the user some control over the features a language-definition file provides. Its first use is for the Greek language, where the user can choose the *πολυτονικό* (“Polutoniko” or multi-accented) Greek way of typesetting texts. These attributes will possibly find wider use in future releases.
- The environment `hyphenrules` is introduced.
- The syntax of the file `language.dat` has been extended to allow (optionally) specifying the font encoding to be used while processing the patterns file.
- The command `\providehyphenmins` should now be used in language definition files in order to be able to keep any settings provided by the pattern file.

## 4.6 Changes in babel version 3.6

In babel version 3.6 a number of bugs that were found in version 3.5 are fixed. Also a number of changes and additions have occurred:

- A new environment `otherlanguage*` is introduced. it only switches the ‘specials’, but leaves the ‘captions’ untouched.
- The shorthands are no longer fully expandable. Some problems could only be solved by peeking at the token following an active character. The advantage is that ‘`{}`’a works as expected for languages that have the ‘ active.
- Support for typesetting french texts is much enhanced; the file `francais.ldf` is now replaced by `frenchb.ldf` which is maintained by Daniel Flipo.
- Support for typesetting the russian language is again available. The language definition file was originally developed by Olga Lapko from CyrTUG. The fonts needed to typeset the russian language are now part of the babel distribution. The support is not yet up to the level which is needed according to Olga, but this is a start.
- Support for typesetting greek texts is now also available. What is offered in this release is a first attempt; it will be enhanced later on by Yannis Haralambous.
- in babel 3.6j some hooks have been added for the development of support for Hebrew typesetting.
- Support for typesetting texts in Afrikaans (a variant of Dutch, spoken in South Africa) has been added to `dutch.ldf`.
- Support for typesetting Welsh texts is now available.
- A new command `\aliasshorthand` is introduced. It seems that in Poland various conventions are used to type the necessary Polish letters. It is now possible to use the character / as a shorthand character instead of the character " , by issuing the command `\aliasshorthand{"}{/}`.
- The shorthand mechanism now deals correctly with characters that are already active.
- Shorthand characters are made active at `\begin{document}`, not earlier. This is to prevent problems with other packages.
- A *preambleonly* command `\substitutefontfamily` has been added to create .fd files on the fly when the font families of the Latin text differ from the families used for the Cyrillic or Greek parts of the text.
- Three new commands `\LdfInit`, `\ldf@quit` and `\ldf@finish` are introduced that perform a number of standard tasks.
- In babel 3.6k the language Ukrainian has been added and the support for Russian typesetting has been adapted to the package ‘cyrillic’ to be released with the December 1998 release of L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>.

## 4.7 Changes in babel version 3.5

In babel version 3.5 a lot of changes have been made when compared with the previous release. Here is a list of the most important ones:

- the selection of the language is delayed until `\begin{document}`, which means you must add appropriate `\selectlanguage` commands if you include `\hyphenation` lists in the preamble of your document.
- babel now has a language environment and a new command `\foreignlanguage`;
- the way active characters are dealt with is completely changed. They are called ‘shorthands’; one can have three levels of shorthands: on the user level, the language level, and on ‘system level’. A consequence of the new way of handling active characters is that they are now written to auxiliary files ‘verbatim’;
- A language change now also writes information in the `.aux` file, as the change might also affect typesetting the table of contents. The consequence is that an `.aux` file generated by a `LATEXformat` with babel preloaded gives errors when read with a `LATEXformat` without babel; but I think this probably doesn’t occur;
- babel is now compatible with the `inputenc` and `fontenc` packages;
- the language definition files now have a new extension, `ldf`;
- the syntax of the file `language.dat` is extended to be compatible with the `french` package by Bernard Gaulle;
- each language definition file looks for a configuration file which has the same name, but the extension `.cfg`. It can contain any valid `LATEX` code.

## 5 Identification and loading of required files

*Code documentation is still under revision.*

The file `babel.sty`<sup>16</sup> is meant for `LATEX 2ε`, therefore we make sure that the format file used is the right one.

The babel installer extends `docstrip` with a few “pseudo-guards” to set “variables” used at installation time. They are used with `<@name@>` at the appropriated places in the source code and shown below with `<<name>>`. That brings a little bit of literate programming.

```
1 <<version=3.9g>>
2 <<date=2013/10/10>>
```

We define some basic which just make the code cleaner. `\bbl@add` is now used internally instead of `\addto` because of the unpredictable behaviour of the latter. Used in `babel.def` and in `babel.sty`, which means in `LATEX` is executed twice, but we need them when defining options and `babel.def` cannot be load until options have been defined.

```
3 <<*Basic macros>> ≡
```

---

<sup>16</sup>The file described in this section is called `babel.dtx`, has version number `v3.9g` and was last revised on 2013/10/10.

```

4 \def\bbl@for#1#2#3{\@for#1:=#2\do{\ifx#1\@empty\else#3\fi}}
5 \def\bbl@add#1#2{%
6   \ifundefined{\expandafter\@gobble\string#1}%
7     {\def#1{#2}}%
8     {\expandafter\def\expandafter#1\expandafter{#1#2}}
9 \def\bbl@csarg#1#2{\expandafter#1\csname bbl@#2\endcsname}%
10 \long\def\bbl@afterelse#1\else#2\fi{\fi#1}
11 \long\def\bbl@afterfi#1\fi{\fi#1}
12 <</Basic macros>>

```

Some files identify themselves with a  $\text{\LaTeX}$  macro. The following code is placed before it to define (and then undefine) if not in  $\text{\LaTeX}$ .

```

13 <<*Make sure ProvidesFile is defined>> ≡
14 \ifx\ProvidesFile\@undefined
15   \def\ProvidesFile#1[#2 #3 #4]{%
16     \wlog{File: #1 #4 #3 <#2>}%
17     \let\ProvidesFile\@undefined}
18 \fi
19 <</Make sure ProvidesFile is defined>>

```

The following code is used in `babel.sty` and `babel.def`, and makes sure the current version of `switch.ldf` is used, if different from that in the format.

```

20 <<*Load switch if newer>> ≡
21 \def\bbl@tempa{<<version>>}%
22 \ifx\bbl@version\bbl@tempa\else
23   \input switch.def\relax
24 \fi
25 <</Load switch if newer>>

```

The following code is used in `babel.def` and `switch.def`.

```

26 <<*Load macros for plain if not LaTeX>> ≡
27 \ifx\AtBeginDocument\@undefined
28   \input plain.def\relax
29 \fi
30 <</Load macros for plain if not LaTeX>>

```

## 5.1 Multiple languages

`\language` Plain  $\text{\TeX}$  version 3.0 provides the primitive `\language` that is used to store the current language. When used with a pre-3.0 version this function has to be implemented by allocating a counter. The following block is used in `switch.def` and `hyphen.cfg`; the latter may seem redundant, but remember `babel` doesn't require loading `switch.def` in the format.

```

31 <<*Define core switching macros>> ≡
32 \ifx\language\@undefined
33   \csname newcount\endcsname\language
34 \fi

```

`\last@language` Another counter is used to store the last language defined. For pre-3.0 formats an extra counter has to be allocated.

`\addlanguage` To add languages to  $\text{\TeX}$ 's memory plain  $\text{\TeX}$  version 3.0 supplies `\newlanguage`, in a pre-3.0 environment a similar macro has to be provided. For both cases a new macro is defined here, because the original `\newlanguage` was defined to be `\outer`.



For a format based on plain version 2.x, the definition of `\newlanguage` can not be copied because `\count 19` is used for other purposes in these formats. Therefore `\addlanguage` is defined using a definition based on the macros used to define `\newlanguage` in plain  $\text{\TeX}$  version 3.0.

For formats based on plain version 3.0 the definition of `\newlanguage` can be simply copied, removing `\outer`.

```

35 \ifx\newlanguage\@undefined
36   \csname newcount\endcsname\last@language
37   \def\addlanguage#1{%
38     \global\advance\last@language\@ne
39     \ifnum\last@language<\@cclvi
40       \else
41         \errmessage{No room for a new \string\language!}%
42       \fi
43     \global\chardef#1\last@language
44     \wlog{\string#1 = \string\language\the\last@language}}

```

Plain  $\text{\TeX}$  version 3.0 uses `\count 19` for this purpose.

```

45 \else
46   \countdef\last@language=19
47   \def\addlanguage{\alloc@9\language\chardef\@cclvi}
48 \fi
49 <</Define core switching macros>>

```

`\ProvidesLanguage` The identification code for each file is something that was introduced in  $\text{\LaTeX} 2_{\epsilon}$ . When the command `\ProvidesFile` does not exist, a dummy definition is provided temporarily. For use in the language definition file the command `\ProvidesLanguage` is defined by babel.

Identify each file that is produced from this source file.

```

50 <{*driver&!user}
51 \ProvidesFile{babel.drv}[<<date>> <<version>>]
52 </driver&!user}
53 <{*driver & user}
54 \ProvidesFile{user.drv}[<<date>> <<version>>]
55 </driver & user}

```

Now we make sure all required files are loaded. When the command `\AtBeginDocument` doesn't exist we assume that we are dealing with a plain-based format or  $\text{\LaTeX} 2.09$ . In that case the file `plain.def` is needed (which also defines `\AtBeginDocument`, and therefore it is not loaded twice). We need the first part when the format is created, and `\orig@dump` is used as a flag. Otherwise, we need to use the second part, so `\orig@dump` is not defined (`plain.def` undefines it). Check if the current version of `switch.def` has been previously loaded (mainly, `hyphen.cfg`). If not, load it now. We cannot load `babel.def` here because we first need to declare and process the package options.

## 6 The Package File

In order to make use of the features of  $\text{\LaTeX} 2_{\epsilon}$ , the babel system contains a package file, `babel.sty`. This file is loaded by the `\usepackage` command and defines all the language options whose name is different from that of the `.ldf` file (like variant spellings). It also takes care of a number of compatibility issues with other packages and defines a few additional package options.

Apart from all the language options below we also have a few options that influence the behaviour of language definition files. Many of the following options don't do anything themselves, they are just defined in order to make it possible for babel and language definition files to check if one of them was specified by the user.

## 6.1 base

The first option to be processed is base, which set the hyphenation patterns then resets ver@babel.sty so that L<sup>A</sup>T<sub>E</sub>X forgets about the first loading. After switch.def has been loaded (above) and \AfterBabelLanguage defined, exits.

```

56 (*package)
57 \ProvidesPackage{babel}[\langle date \rangle \langle version \rangle] The Babel package]
58 \@ifpackagewith{babel}{debug}
59 {\input switch.def\relax}
60 {\langle Load switch if newer \rangle}
61 \langle Basic macros \rangle
62 \def\AfterBabelLanguage#1{%
63   \global\expandafter\bbledadd\csname#1.ldf-h@@k\endcsname}%
64 \@ifpackagewith{babel}{base}{%
65   \DeclareOption*{\bbledpatterns{\CurrentOption}}%
66   \DeclareOption{base}{}%
67   \ProcessOptions
68   \global\expandafter\let\csname opt@babel.sty\endcsname\relax
69   \global\expandafter\let\csname ver@babel.sty\endcsname\relax
70   \global\let\@ifl@ter@@\@ifl@ter
71   \def\@ifl@ter#1#2#3#4#5{\global\let\@ifl@ter\@ifl@ter@@}%
72   \endinput}{}%

```

## 6.2 key=value options and other general option

The following macros extract language modifiers, and only real package options are kept in the option list. Modifiers are saved and assigned to \BabelModifiers at \bbledload@language; when no modifiers have been given, the former is \relax. How modifiers are handled are left to language styles; they can use \in@, loop them with \@for o load keyval).

```

73 \bbledcsarg\let{tempa\expandafter}\csname opt@babel.sty\endcsname
74 \def\bbledtempb#1.#2{%
75   #1\ifx\@empty#2\else,\bbledafterfi\bbledtempb#2\fi}%
76 \def\bbledtempd#1.#2\@nnil{%
77   \ifx\@empty#2%
78     \edef\bbledtempc{\ifx\bbledtempc\@empty\else\bbledtempc,\fi#1}%
79   \else
80     \in@{=}{#1}\ifin@
81     \edef\bbledtempc{\ifx\bbledtempc\@empty\else\bbledtempc,\fi#1.#2}%
82   \else
83     \edef\bbledtempc{\ifx\bbledtempc\@empty\else\bbledtempc,\fi#1}%
84     \bbledcsarg\edef{mod@#1}{\bbledtempb#2}%
85   \fi
86 \fi}
87 \let\bbledtempc\@empty
88 \bbledfor\bbledtempa\bbledtempa{%
89   \expandafter\bbledtempd\bbledtempa.\@empty\@nnil}
90 \expandafter\let\csname opt@babel.sty\endcsname\bbledtempc

```

```

91 \DeclareOption{activeacute}{}
92 \DeclareOption{activegrave}{}

```

The next option tells babel to leave shorthand characters active at the end of processing the package. This is *not* the default as it can cause problems with other packages, but for those who want to use the shorthand characters in the preamble of their documents this can help.

```

93 \DeclareOption{KeepShorthandsActive}{}
94 \DeclareOption{debug}{}
95 \DeclareOption{noconfigs}{}
96 \DeclareOption{showlanguages}{}
97 \DeclareOption{shorthands=off}{\bbl@tempa shorthands=\bbl@tempa}
98 \DeclareOption{strings}{\def\bbl@opt@strings{\BabelStringsDefault}}
99 \DeclareOption{strings=encoded}{\let\bbl@opt@strings\relax}
100 \def\BabelStringsDefault{generic}
101 \DeclareOption{safe=none}{\let\bbl@opt@safe\@empty}
102 \DeclareOption{safe=bib}{\def\bbl@opt@safe{B}}
103 \DeclareOption{safe=ref}{\def\bbl@opt@safe{R}}
104 \DeclareOption{math=active}{}
105 \DeclareOption{math=normal}{\def\bbl@mathnormal{\noexpand\textormath}}
106 \DeclareOption{hyphenmap=off}{\chardef\bbl@hymapopt\z@}
107 \DeclareOption{hyphenmap=first}{\chardef\bbl@hymapopt\@ne}
108 \DeclareOption{hyphenmap=select}{\chardef\bbl@hymapopt\tw@}
109 \DeclareOption{hyphenmap=other}{\chardef\bbl@hymapopt\thr@}
110 \DeclareOption{hyphenmap=other*}{\chardef\bbl@hymapopt4\relax}

```

Handling of package options is done in three passes. (I [JBL] am not very happy with the idea, anyway.) The first one processes options which has been declared above or follow the syntax `<key>=<value>`, the second one loads the requested languages, except the main one if set with the key `main`, and the third one loads the latter. First, we “flag” valid keys with a nil value.

```

111 \let\bbl@opt@shorthands\@nnil
112 \let\bbl@opt@config\@nnil
113 \let\bbl@opt@main\@nnil
114 \let\bbl@opt@strings\@nnil
115 \let\bbl@opt@headfoot\@nnil

```

The following tool is defined temporarily to store the values of options.

```

116 \def\bbl@tempa#1=#2\bbl@tempa{%
117   \expandafter\ifx\csname bbl@opt@#1\endcsname\@nnil
118     \expandafter\edef\csname bbl@opt@#1\endcsname{#2}%
119   \else
120     \bbl@error{%
121       Bad option ‘#1=#2’. Either you have misspelled the\\%
122       key or there is a previous setting of ‘#1’}%
123     Valid keys are ‘shorthands’, ‘config’, ‘strings’, ‘main’,\\%
124     ‘headfoot’, ‘safe’, ‘math’}
125   \fi}

```

Now the option list is processed, taking into account only currently declared options (including those declared with a `=`), and `<key>=<value>` options (the former take precedence). Unrecognized options are saved in `\bbl@language@opts`, because they are language options.

```

126 \let\bbl@language@opts\@empty
127 \DeclareOption*{%
128   \@expandtwoargs\in@{\string=}{\CurrentOption}%
129   \ifin@

```

```

130 \expandafter\bb@tempa\CurrentOption\bb@tempa
131 \else
132 \edef\bb@language@opts{%
133 \ifx\bb@language@opts\empty\else\bb@language@opts,\fi
134 \CurrentOption}%
135 \fi}

```

Now we finish the first pass (and start over).

```

136 \ProcessOptions*

```

### 6.3 Conditional loading of shorthands

If there is no `shorthands=<chars>`, the original babel macros are left untouched, but if there is, these macros are wrapped (in `babel.def`) to define only those given. A bit of optimization: if there is no `shorthands=`, then `\bb@ifshorthands` is always true, and it is always false if `shorthands` is empty. Also, some code makes sense only with `shorthands=...`

```

137 \def\bb@sh@string#1{%
138 \ifx#1\empty\else
139 \ifx#1t\string~%
140 \else\ifx#1c\string,%
141 \else\string#1%
142 \fi\fi
143 \expandafter\bb@sh@string
144 \fi}
145 \ifx\bb@opt@shorthands\@nnil
146 \def\bb@ifshorthand#1#2#3{#2}%
147 \else\ifx\bb@opt@shorthands\empty
148 \def\bb@ifshorthand#1#2#3{#3}%
149 \else

```

The following macro tests if a shorthand is one of the allowed ones.

```

150 \def\bb@ifshorthand#1{%
151 \@expandtwoargs\in{\string#1}{\bb@opt@shorthands}%
152 \ifin@
153 \expandafter\@firstoftwo
154 \else
155 \expandafter\@secondoftwo
156 \fi}

```

We make sure all chars in the string are ‘other’, with the help of an auxiliary macro defined above (which also zaps spaces).

```

157 \edef\bb@opt@shorthands{%
158 \expandafter\bb@sh@string\bb@opt@shorthands\empty}%

```

The following is ignored with `shorthands=off`, since it is intended to take some additional actions for certain chars.

```

159 \bb@ifshorthand{'}%
160 {\PassOptionsToPackage{activeacute}{babel}}{}
161 \bb@ifshorthand{'}%
162 {\PassOptionsToPackage{activegrave}{babel}}{}
163 \fi\fi

```

With `headfoot=lang` we can set the language used in heads/foots. For example, in `babel/3796` just adds `headfoot=english`. It misuses `\@resetactivechars` but seems to work.

```

164 \ifx\bb@opt@headfoot\@nnil\else

```

```

165 \g@addto@macro\@resetactivechars{%
166   \set@typeset@protect
167   \expandafter\select@language@x\expandafter{\bbl@opt@headfoot}%
168   \let\protect\noexpand}
169 \fi

```

For the option `safe` we use a different approach – `\bbl@opt@safe` says which macros are redefined (B for bibs and R for refs). By default, both are set.

```

170 \ifundefined{bbl@opt@safe}{\def\bbl@opt@safe{BR}}{}
171 \ifx\bbl@opt@main\@nnil\else
172   \edef\bbl@language@opts{%
173     \ifx\bbl@language@opts\@empty\else\bbl@language@opts,\fi
174     \bbl@opt@main}
175 \fi

```

If the format created a list of loaded languages (in `\bbl@languages`), get the name of the 0-th to show the actual language used.

```

176 \ifx\bbl@languages\@undefined\else
177   \begingroup
178   \catcode'\^^I=12
179   \@ifpackagewith{babel}{showlanguages}{%
180     \begingroup
181       \def\bbl@elt#1#2#3#4{\wlog{#2^^I#1^^I#3^^I#4}}%
182       \wlog{< *languages >}%
183       \bbl@languages
184       \wlog{< /languages >}%
185     \endgroup}{}
186   \endgroup
187   \def\bbl@elt#1#2#3#4{%
188     \ifnum#2=\z@
189       \gdef\bbl@nulllanguage{#1}%
190       \def\bbl@elt##1##2##3##4{}%
191     \fi}%
192   \bbl@languages
193 \fi

```

## 6.4 Language options

Languages are loaded when processing the corresponding option *except* if a main language has been set. In such a case, it is not loaded until all options has been processed. The following macro inputs the `ldf` file and does some additional checks (`\input` works, too, but possible errors are not caught).

```

194 \let\bbl@afterlang\relax
195 \let\BabelModifiers\relax
196 \let\bbl@loaded\@empty
197 \def\bbl@load@language#1{%
198   \InputIfFileExists{#1.ldf}%
199   {\edef\bbl@loaded{\CurrentOption
200     \ifx\bbl@loaded\@empty\else,\bbl@loaded\fi}%
201     \expandafter\let\expandafter\bbl@afterlang
202     \csname\CurrentOption.ldf-h@k\endcsname
203     \expandafter\let\expandafter\BabelModifiers
204     \csname bbl@mod@\CurrentOption\endcsname}%
205   {\bbl@error{%
206     Unknown option '\CurrentOption'. Either you misspelled it\\%
207     or the language definition file \CurrentOption.ldf was not found}}%

```

```

208 Valid options are: shorthands=, KeepShorthandsActive,\%
209 activeacute, activegrave, noconfigs, safe=, main=, math=\%
210 headfoot=, strings=, config=, hyphenmap=, or a language name.}}

```

Now, we set language options whose names are different from ldf files.

```

211 \DeclareOption{acadian}{\bbl@load@language{frenchb}}
212 \DeclareOption{afrikaans}{\bbl@load@language{dutch}}
213 \DeclareOption{american}{\bbl@load@language{english}}
214 \DeclareOption{australian}{\bbl@load@language{english}}
215 \DeclareOption{austrian}{\bbl@load@language{germanb}}
216 \DeclareOption{bahasa}{\bbl@load@language{bahasai}}
217 \DeclareOption{bahasai}{\bbl@load@language{bahasai}}
218 \DeclareOption{bahasam}{\bbl@load@language{bahasam}}
219 \DeclareOption{brazil}{\bbl@load@language{portuges}}
220 \DeclareOption{brazilian}{\bbl@load@language{portuges}}
221 \DeclareOption{british}{\bbl@load@language{english}}
222 \DeclareOption{canadian}{\bbl@load@language{english}}
223 \DeclareOption{canadien}{\bbl@load@language{frenchb}}
224 \DeclareOption{francais}{\bbl@load@language{frenchb}}
225 \DeclareOption{french}{\bbl@load@language{frenchb}}%
226 \DeclareOption{german}{\bbl@load@language{germanb}}
227 \DeclareOption{hebrew}{%
228   \input{rlbabel.def}%
229   \bbl@load@language{hebrew}}
230 \DeclareOption{hungarian}{\bbl@load@language{magyar}}
231 \DeclareOption{indon}{\bbl@load@language{bahasai}}
232 \DeclareOption{indonesian}{\bbl@load@language{bahasai}}
233 \DeclareOption{lowersorbian}{\bbl@load@language{lsorbian}}
234 \DeclareOption{malay}{\bbl@load@language{bahasam}}
235 \DeclareOption{meyalu}{\bbl@load@language{bahasam}}
236 \DeclareOption{naustrian}{\bbl@load@language{ngermanb}}
237 \DeclareOption{newzealand}{\bbl@load@language{english}}
238 \DeclareOption{ngerman}{\bbl@load@language{ngermanb}}
239 \DeclareOption{nynorsk}{\bbl@load@language{norsk}}
240 \DeclareOption{polutonikogreek}{%
241   \bbl@load@language{greek}%
242   \languageattribute{greek}{polutoniko}}
243 \DeclareOption{portuguese}{\bbl@load@language{portuges}}
244 \DeclareOption{russian}{\bbl@load@language{russianb}}
245 \DeclareOption{UKenglish}{\bbl@load@language{english}}
246 \DeclareOption{ukrainian}{\bbl@load@language{ukraineb}}
247 \DeclareOption{uppersorbian}{\bbl@load@language{usorbian}}
248 \DeclareOption{USenglish}{\bbl@load@language{english}}

```

Another way to extend the list of ‘known’ options for babel is to create the file `bblopts.cfg` in which one can add option declarations. However, this mechanism is deprecated – if you want an alternative name for a language, just create a new `.ldf` file loading the actual one. You can also set the name of the file with the package option `config=<name>`, which will load `<name>.cfg` instead.

```

249 \ifx\bbl@opt@config\annil
250   \ifpackagewith{babel}{noconfigs}}}%
251   {\InputIfFileExists{bblopts.cfg}%
252     {\typeout{*****^J%
253               * Local config file bblopts.cfg used^J%
254               *}}}%
255     {}}}%
256 \else

```

```

257 \InputIfFileExists{\bbl@opt@config.cfg}%
258 {\typeout{*****^J}%
259      * Local config file \bbl@opt@config.cfg used^J%
260      *}}%
261 {\bbl@error{%
262   Local config file '\bbl@opt@config.cfg' not found}{%
263   Perhaps you misspelled it.}}%
264 \fi

```

Recognizing global options in packages not having a closed set of them is not trivial, as for them to be processed they must be defined explicitly. So, package options not yet taken into account and stored in `bbl@language@opts` are assumed to be languages (note this list also contains the language given with `main`). If not declared above, the name of the option and the file are the same.

```

265 \bbl@for\bbl@tempa\bbl@language@opts{%
266   \ifundefined{ds@\bbl@tempa}%
267   {\edef\bbl@tempb{%
268     \noexpand\DeclareOption
269     {\bbl@tempa}%
270     {\noexpand\bbl@load@language{\bbl@tempa}}}%
271    \bbl@tempb}%
272   \@empty}

```

Now, we make sure an option is explicitly declared for any language set as global option, by checking if an `ldf` exists. The previous step was, in fact, somewhat redundant, but that way we minimize accessing the file system just to see if the option could be a language.

```

273 \bbl@for\bbl@tempa\@classoptionslist{%
274   \ifundefined{ds@\bbl@tempa}%
275   {\IfFileExists{\bbl@tempa.ldf}%
276    {\edef\bbl@tempb{%
277      \noexpand\DeclareOption
278      {\bbl@tempa}%
279      {\noexpand\bbl@load@language{\bbl@tempa}}}%
280     \bbl@tempb}%
281    \@empty}%
282   \@empty}

```

If a main language has been set, store it for the third pass.

```

283 \ifx\bbl@opt@main\@nnil\else
284   \expandafter
285   \let\expandafter\bbl@loadmain\csname ds@\bbl@opt@main\endcsname
286   \DeclareOption{\bbl@opt@main}{}
287 \fi

```

And we are done, because all options for this pass has been declared. Those already processed in the first pass are just ignored.

The options have to be processed in the order in which the user specified them (except, of course, global options, which `LATEX` processes before):

```

288 \def\AfterBabelLanguage#1{%
289   \bbl@ifsamestring\CurrentOption{#1}{\global\bbl@add\bbl@afterlang{}}
290   \DeclareOption*{}
291 \ProcessOptions*

```

This finished the second pass. Now the third one begins, which loads the main language set with the key `main`. A warning is raised if the main language is not the same as the last named one, or if the value of the key `main` is not a language. Then

execute directly the option (because it could be used only in main). After loading all languages, we deactivate \AfterBabelLanguage.

```

292 \ifx\bbl@opt@main\@nnil
293 \edef\bbl@tempa{\@classoptionslist,\bbl@language@opts}
294 \let\bbl@tempc\@empty
295 \bbl@for\bbl@tempb\bbl@tempa{%
296   \@expandtwoargs\in{,\bbl@tempb,}{,\bbl@loaded,}%
297   \ifin\edef\bbl@tempc{\bbl@tempb}\fi}
298 \def\bbl@tempa#1,#2\@nnil{\def\bbl@tempb{#1}}
299 \expandafter\bbl@tempa\bbl@loaded,\@nnil
300 \ifx\bbl@tempb\bbl@tempc\else
301   \bbl@warning{%
302     Last declared language option is '\bbl@tempc,\,%
303     but the last processed one was '\bbl@tempb'.\,%
304     The main language cannot be set as both a global\,%
305     and a package option. Use 'main=\bbl@tempc' as\,%
306     option. Reported}%
307   \fi
308 \else
309   \DeclareOption{\bbl@opt@main}{\bbl@loadmain}
310   \ExecuteOptions{\bbl@opt@main}
311   \DeclareOption*{}
312   \ProcessOptions*
313 \fi
314 \def\AfterBabelLanguage{%
315   \bbl@error
316   {Too late for \string\AfterBabelLanguage}%
317   {Languages have been loaded, so I can do nothing}}
318 \ifx\bbl@main@language\@undefined
319   \bbl@error{%
320     You haven't specified a language option}{%
321     You need to specify a language, either as a global option\,%
322     or as an optional argument to the \string\usepackage\space
323     command;\,%
324     You shouldn't try to proceed from here, type x to quit.}
325 \fi
326 \end{package}

```

## 7 The Kernel of Babel

The kernel of the babel system is stored in either hyphen.cfg or switch.def and babel.def. The file babel.def contains most of the code, while switch.def defines the language switching commands; both can be read at run time. The file hyphen.cfg is a file that can be loaded into the format, which is necessary when you want to be able to switch hyphenation patterns (by default, it also inputs switch.def, for “historical reasons”, but it is not necessary). When babel.def is loaded it checks if the current version of switch.def is in the format; if not it is loaded. A further file, babel.sty, contains L<sup>A</sup>T<sub>E</sub>X-specific stuff.

Because plain T<sub>E</sub>X users might want to use some of the features of the babel system too, care has to be taken that plain T<sub>E</sub>X can process the files. For this



reason the current format will have to be checked in a number of places. Some of the code below is common to plain  $\text{\TeX}$  and  $\text{\LaTeX}$ , some of it is for the  $\text{\LaTeX}$  case only.

## 7.1 Tools

`\bbl@engine` takes the following values: 0 is  $\text{\pdfTeX}$ , 1 is  $\text{\luatex}$ , and 2 is  $\text{\xetex}$ . You may use it in your language style if necessary.

```

327 <{*core}>
328 <<Make sure ProvidesFile is defined>>
329 \ProvidesFile{babel.def}[\<date>] [\<version>] Babel common definitions]
330 <<Load macros for plain if not LaTeX>>
331 <<Load switch if newer>>
332 <<Basic macros>>
333 \ifx\bbl@opt@strings\@undefined
334   \def\bbl@opt@safe{BR}
335   \let\bbl@opt@strings\@nnil
336   \let\bbl@opt@shorthands\@nnil
337 \fi
338 \ifx\BabelModifiers\@undefined\let\BabelModifiers\relax\fi
339 \ifx\bbl@afterlang\@undefined\let\bbl@afterlang\relax\fi
340 \providecommand\AfterBabelLanguage[2]{}
341 \def\bbl@csarg#1#2{\expandafter#1\csname bbl@#2\endcsname}%
342 \chardef\bbl@engine=%
343   \ifx\directlua\@undefined
344     \ifx\XeTeXinputencoding\@undefined
345       \z@
346     \else
347       \tw@
348     \fi
349   \else
350     \@ne
351   \fi

```

`\bbl@afterelse`    Because the code that is used in the handling of active characters may need to look ahead, we take extra care to ‘throw’ it over the `\else` and `\fi` parts of an `\if-statement`<sup>17</sup>. These macros will break if another `\if... \fi` statement appears in one of the arguments and it is not enclosed in braces.

```

352 \long\def\bbl@afterelse#1\else#2\fi{\fi#1}
353 \long\def\bbl@afterfi#1\fi{\fi#1}

```

The macro `\initiate@active@char` takes all the necessary actions to make its argument a shorthand character. The real work is performed once for each character.

```

354 \def\bbl@withactive#1#2{%
355   \begingroup
356   \lccode'\sim'#2\relax
357   \lowercase{\endgroup#1~}}

```

`\bbl@redefine`    To redefine a command, we save the old meaning of the macro. Then we redefine it to call the original macro with the ‘sanitized’ argument. The reason why we do it

---

<sup>17</sup>This code is based on code presented in TUGboat vol. 12, no2, June 1991 in “An expansion Power Lemma” by Sonja Maus.

this way is that we don't want to redefine the L<sup>A</sup>T<sub>E</sub>X macros completely in case their definitions change (they have changed in the past).

Because we need to redefine a number of commands we define the command `\bbl@redefine` which takes care of this. It creates a new control sequence, `\org@...`

```
358 \def\bbl@redefine#1{%
359   \edef\bbl@tempa{\expandafter\@gobble\string#1}%
360   \expandafter\let\csname org@\bbl@tempa\endcsname#1%
361   \expandafter\def\csname\bbl@tempa\endcsname}
```

This command should only be used in the preamble of the document.

```
362 \@onlypreamble\bbl@redefine
```

`\bbl@redefine@long` This version of `\babel@redefine` can be used to redefine `\long` commands such as `\ifthenelse`.

```
363 \def\bbl@redefine@long#1{%
364   \edef\bbl@tempa{\expandafter\@gobble\string#1}%
365   \expandafter\let\csname org@\bbl@tempa\endcsname#1%
366   \expandafter\long\expandafter\def\csname\bbl@tempa\endcsname}
367 \@onlypreamble\bbl@redefine@long
```

`\bbl@redefineroobust` For commands that are redefined, but which *might* be robust we need a slightly more intelligent macro. A robust command `foo` is defined to expand to `\protect\foo_`. So it is necessary to check whether `\foo_` exists. The result is that the command that is being redefined is always robust afterwards. Therefore all we need to do now is define `\foo_`.

```
368 \def\bbl@redefineroobust#1{%
369   \edef\bbl@tempa{\expandafter\@gobble\string#1}%
370   \expandafter\ifx\csname\bbl@tempa\space\endcsname\relax
371     \expandafter\let\csname org@\bbl@tempa\endcsname#1%
372     \expandafter\edef\csname\bbl@tempa\endcsname{\noexpand\protect
373       \expandafter\noexpand\csname\bbl@tempa\space\endcsname}%
374   \else
375     \expandafter\let\csname org@\bbl@tempa\expandafter\endcsname
376       \csname\bbl@tempa\space\endcsname
377   \fi
378   \expandafter\def\csname\bbl@tempa\space\endcsname}
```

This command should only be used in the preamble of the document.

```
379 \@onlypreamble\bbl@redefineroobust
```

## 7.2 Encoding issues

The first thing we need to do is to determine, at `\begin{document}`, which latin fontencoding to use.

`\latinencoding` When text is being typeset in an encoding other than 'latin' (OT1 or T1), it would be nice to still have Roman numerals come out in the Latin encoding. So we first assume that the current encoding at the end of processing the package is the Latin encoding.

```
380 \AtEndOfPackage{\edef\latinencoding{\cf@encoding}}
```

But this might be overruled with a later loading of the package `fontenc`. Therefore we check at the execution of `\begin{document}` whether it was loaded with the T1 option. The normal way to do this (using `\@ifpackageloaded`) is disabled for this

package. Now we have to revert to parsing the internal macro `\@filelist` which contains all the filenames loaded.

```

381 \AtBeginDocument{%
382   \gdef\latinencoding{OT1}%
383   \ifx\cf@encoding\bbl@t@one
384     \xdef\latinencoding{\bbl@t@one}%
385   \else
386     \@ifl@aded{def}{tlenc}{\xdef\latinencoding{\bbl@t@one}}{}%
387   \fi
388 }
```

`\latintext` Then we can define the command `\latintext` which is a declarative switch to a latin font-encoding.

```

389 \DeclareRobustCommand{\latintext}{%
390   \fontencoding{\latinencoding}\selectfont
391   \def\encodingdefault{\latinencoding}}
```

`\textlatin` This command takes an argument which is then typeset using the requested font encoding. In order to avoid many encoding switches it operates in a local scope.

```

392 \ifx\@undefined\DeclareTextFontCommand
393   \DeclareRobustCommand{\textlatin}[1]{\leavevmode{\latintext #1}}
394 \else
395   \DeclareTextFontCommand{\textlatin}{\latintext}
396 \fi
```

The second version of this macro takes two arguments. The first argument is the name of the language that will be defined in the language definition file; the second argument is either a control sequence or a string from which a control sequence should be constructed. The existence of the control sequence indicates that the file has been processed before.

At the start of processing a language definition file we always check the category code of the at-sign. We make sure that it is a ‘letter’ during the processing of the file. We also save its name as the last called option, even if not loaded.

Another character that needs to have the correct category code during processing of language definition files is the equals sign, ‘=’, because it is sometimes used in constructions with the `\let` primitive. Therefore we store its current catcode and restore it later on.

Now we check whether we should perhaps stop the processing of this file. To do this we first need to check whether the second argument that is passed to `\LdfInit` is a control sequence. We do that by looking at the first token after passing #2 through `string`. When it is equal to `\@backslashchar` we are dealing with a control sequence which we can compare with `\@undefined`.

If so, we call `\ldf@quit` to set the main language, restore the category code of the @-sign and call `\endinput`

When #2 was *not* a control sequence we construct one and compare it with `\relax`. Finally we check `\originalTeX`.

```

397 \def\LdfInit#1#2{%
398   \chardef\atcatcode=\catcode'\@
399   \catcode'\@=11\relax
400   \chardef\eqcatcode=\catcode'\=
401   \catcode'\==12\relax
402   \expandafter\if\expandafter\@backslashchar
403     \expandafter\@car\string#2\@nil
404   \ifx#2\@undefined\else
```

```

405     \ldf@quit{#1}%
406     \fi
407 \else
408     \expandafter\ifx\csname#2\endcsname\relax\else
409     \ldf@quit{#1}%
410     \fi
411 \fi
412 \let\bbl@screset\@empty
413 \let\BabelStrings\bbl@opt@strings
414 \let\BabelOptions\@empty
415 \let\BabelLanguages\relax
416 \ifx\originalTeX\@undefined
417     \let\originalTeX\@empty
418 \else
419     \originalTeX
420 \fi}

```

`\ldf@quit` This macro interrupts the processing of a language definition file.

```

421 \def\ldf@quit#1{%
422     \expandafter\main@language\expandafter{#1}%
423     \catcode'\@=\atcatcode \let\atcatcode\relax
424     \catcode'\==\eqcatcode \let\eqcatcode\relax
425     \endinput}

```

`\ldf@finish` This macro takes one argument. It is the name of the language that was defined in the language definition file.

We load the local configuration file if one is present, we set the main language (taking into account that the argument might be a control sequence that needs to be expanded) and reset the category code of the @-sign.

```

426 \def\ldf@finish#1{%
427     \loadlocalcfg{#1}%
428     \bbl@afterlang
429     \let\bbl@afterlang\relax
430     \let\BabelModifiers\relax
431     \let\bbl@screset\relax
432     \expandafter\main@language\expandafter{#1}%
433     \catcode'\@=\atcatcode \let\atcatcode\relax
434     \catcode'\==\eqcatcode \let\eqcatcode\relax}

```

After the preamble of the document the commands `\LdfInit`, `\ldf@quit` and `\ldf@finish` are no longer needed. Therefore they are turned into warning messages in *L<sup>A</sup>T<sub>E</sub>X*.

```

435 \@onlypreamble\LdfInit
436 \@onlypreamble\ldf@quit
437 \@onlypreamble\ldf@finish

```

`\main@language` This command should be used in the various language definition files. It stores its argument in `\bbl@main@language`; to be used to switch to the correct language at the beginning of the document.

```

438 \def\main@language#1{%
439     \def\bbl@main@language{#1}%
440     \let\language\name\bbl@main@language
441     \bbl@patterns{\language}}

```

We also have to make sure that some code gets executed at the beginning of the document.

```

442 \AtBeginDocument{%
443   \expandafter\selectlanguage\expandafter{\bbl@main@language}}

```

### 7.3 Support for active characters

`\bbl@add@special` The macro `\bbl@add@special` is used to add a new character (or single character control sequence) to the macro `\dospecials` (and `\@sanitize` if L<sup>A</sup>T<sub>E</sub>X is used). To keep all changes local, we begin a new group. Then we redefine the macros `\do` and `\makeother` to add themselves and the given character without expansion. To add the character to the macros, we expand the original macros with the additional character inside the redefinition of the macros. Because `\@sanitize` can be undefined, we put the definition inside a conditional.

```

444 \def\bbl@add@special#1{%
445   \begingroup
446     \def\do{\noexpand\do\noexpand}%
447     \def\makeother{\noexpand\makeother\noexpand}%
448   \edef\x{\endgroup
449     \def\noexpand\dospecials{\dospecials\do#1}%
450     \expandafter\ifx\csname @sanitize\endcsname\relax \else
451       \def\noexpand\@sanitize{\@sanitize\makeother#1}%
452     \fi}%
453   \x}

```

The macro `\x` contains at this moment the following:

`\endgroup\def\dospecials{old contents \do<char>}`.

If `\@sanitize` is defined, it contains an additional definition of this macro. The last thing we have to do, is the expansion of `\x`. Then `\endgroup` is executed, which restores the old meaning of `\x`, `\do` and `\makeother`. After the group is closed, the new definition of `\dospecials` (and `\@sanitize`) is assigned.

`\bbl@remove@special` The companion of the former macro is `\bbl@remove@special`. It is used to remove a character from the set macros `\dospecials` and `\@sanitize`.

To keep all changes local, we begin a new group. Then we define a help macro `\x`, which expands to empty if the characters match, otherwise it expands to its nonexpandable input. Because T<sub>E</sub>X inserts a `\relax`, if the corresponding `\else` or `\fi` is scanned before the comparison is evaluated, we provide a ‘stop sign’ which should expand to nothing.

With the help of this macro we define `\do` and `\makeother`.

The rest of the work is similar to `\bbl@add@special`.

```

454 \def\bbl@remove@special#1{%
455   \begingroup
456     \def\x##1##2{\ifnum'#1='##2\noexpand\@empty
457       \else\noexpand##1\noexpand##2\fi}%
458     \def\do{\x\do}%
459     \def\makeother{\x\makeother}%
460   \edef\x{\endgroup
461     \def\noexpand\dospecials{\dospecials}%
462     \expandafter\ifx\csname @sanitize\endcsname\relax \else
463       \def\noexpand\@sanitize{\@sanitize}%
464     \fi}%
465   \x}

```

## 7.4 Shorthands

`\initiate@active@char` A language definition file can call this macro to make a character active. This macro takes one argument, the character that is to be made active. When the character was already active this macro does nothing. Otherwise, this macro defines the control sequence `\normal@char⟨char⟩` to expand to the character in its ‘normal state’ and it defines the active character to expand to `\normal@char⟨char⟩` by default (`⟨char⟩` being the character to be made active). Later its definition can be changed to expand to `\active@char⟨char⟩` by calling `\bbl@activate{⟨char⟩}`. For example, to make the double quote character active one could have `\initiate@active@char{"}` in a language definition file. This defines `"` as `\active@prefix "\active@char"` (where the first `"` is the character with its original catcode, when the shorthand is created, and `\active@char` is a single token). In protected contexts, it expands to `\protect "` or `\noexpand "` (ie, with the original `"`); otherwise `\active@char` is executed. This macro in turn expands to `\normal@char` in “safe” contexts (eg, `\label`), but `\user@active` in normal “unsafe” ones. The latter search a definition in the user, language and system levels, in this order, but if none is found, `\normal@char` is used. However, a deactivated shorthand (with `\bbl@deactivate` is defined as `\active@prefix "\normal@char"`. The following macro is used to define shorthands in the three levels. It takes 4 arguments: the (string’ed) character, `<level>@group`, `<level>@active` and `<next-level>@active` (except in system).

```

466 \def\bbl@active@def#1#2#3#4{%
467   \@namedef{#3#1}{%
468     \expandafter\ifx\csname#2@sh@#1\endcsname\relax
469     \bbl@afterelse\bbl@sh@select#2#1{#3@arg#1}{#4#1}%
470   \else
471     \bbl@afterfi\csname#2@sh@#1\endcsname
472   \fi}%

```

When there is also no current-level shorthand with an argument we will check whether there is a next-level defined shorthand for this active character.

```

473   \long\@namedef{#3@arg#1}##1{%
474     \expandafter\ifx\csname#2@sh@#1\string##1\endcsname\relax
475     \bbl@afterelse\csname#4#1\endcsname##1%
476   \else
477     \bbl@afterfi\csname#2@sh@#1\string##1\endcsname
478   \fi}}%

```

`\initiate@active@char` calls `\@initiate@active@char` with 3 arguments. All of them are the same character with different catcodes: active, other (string’ed) and the original one.

```

479 \def\initiate@active@char#1{%
480   \expandafter\ifx\csname active@char\string#1\endcsname\relax
481   \bbl@withactive
482   {\expandafter\@initiate@active@char\expandafter}#1\string#1#1%
483   \fi}

```

The very first thing to do is saving the original catcode and the original definition, even if not active, which is possible (undefined characters require a special treatment to avoid making them `\relax`).

```

484 \def\@initiate@active@char#1#2#3{%
485   \expandafter\edef\csname bbl@oricat@#2\endcsname{%
486     \catcode'#2=\the\catcode'#2\relax}%

```

```

487 \ifx#1\undefined
488   \expandafter\edef\csname bbl@oridef@#2\endcsname{%
489     \let\noexpand#1\noexpand\undefined}%
490 \else
491   \expandafter\let\csname bbl@oridef@#2\endcsname#1%
492   \expandafter\edef\csname bbl@oridef@#2\endcsname{%
493     \let\noexpand#1%
494     \expandafter\noexpand\csname bbl@oridef@#2\endcsname}%
495 \fi

```

If the character is already active we provide the default expansion under this shorthand mechanism. Otherwise we write a message in the transcript file, and define `\normal@char⟨char⟩` to expand to the character in its default state. If the character is mathematically active when babel is loaded (for example `'`) the normal expansion is somewhat different to avoid an infinite loop (but it does not prevent the loop if the mathcode is set to `"8000 a posteriori`).

```

496 \ifx#1#3\relax
497   \expandafter\let\csname normal@char#2\endcsname#3%
498 \else
499   \bbl@info{Making #2 an active character}%
500   \ifnum\mathcode'#2="8000
501     \@namedef{normal@char#2}{%
502       \textormath{#3}{\csname bbl@oridef@#2\endcsname}}%
503   \else
504     \@namedef{normal@char#2}{#3}%
505 \fi

```

To prevent problems with the loading of other packages after babel we reset the catcode of the character to the original one at the end of the package and of each language file (except with `KeepShorthandsActive`). It is re-activate again at `\begin{document}`. We also need to make sure that the shorthands are active during the processing of the `.aux` file. Otherwise some citations may give unexpected results in the printout when a shorthand was used in the optional argument of `\bibitem` for example. . Then we make it active (not strictly necessary, but done for backward compatibility).

```

506   \bbl@restoreactive{#2}%
507   \AtBeginDocument{%
508     \catcode'#2\active
509     \if@filesw
510       \immediate\write\@mainaux{\catcode'\string#2\active}%
511     \fi}%
512   \expandafter\bbl@add@special\csname#2\endcsname
513   \catcode'#2\active
514 \fi

```

Now we have set `\normal@char⟨char⟩`, we must define `\active@char⟨char⟩`, to be executed when the character is activated. We define the first level expansion of `\active@char⟨char⟩` to check the status of the `@safe@actives` flag. If it is set to true we expand to the `'normal'` version of this character, otherwise we call `\user@active⟨char⟩` to start the search of a definition in the user, language and system levels (or eventually `normal@char⟨char⟩`).

```

515 \let\bbl@tempa\@firstoftwo
516 \if\string^#2%
517   \def\bbl@tempa{\noexpand\textormath}%
518 \else
519   \ifx\bbl@mathnormal\undefined\else

```

```

520     \let\bbl@tempa\bbl@mathnormal
521     \fi
522 \fi
523 \expandafter\edef\csname active@char#2\endcsname{%
524     \bbl@tempa
525     {\noexpand\if@safe@actives
526       \noexpand\expandafter
527       \expandafter\noexpand\csname normal@char#2\endcsname
528       \noexpand\else
529       \noexpand\expandafter
530       \expandafter\noexpand\csname user@active#2\endcsname
531       \noexpand\fi}%
532     {\expandafter\noexpand\csname normal@char#2\endcsname}}%

```

We now define the default values which the shorthand is set to when activated or deactivated. It is set to the deactivated form (globally), so that the character expands to

`\active@prefix <char> \normal@char<char>`

(where `\active@char<char>` is *one* control sequence!).

```

533 \bbl@csarg\edef{active@#2}{%
534     \noexpand\active@prefix\noexpand#1%
535     \expandafter\noexpand\csname active@char#2\endcsname}%
536 \bbl@csarg\edef{normal@#2}{%
537     \noexpand\active@prefix\noexpand#1%
538     \expandafter\noexpand\csname normal@char#2\endcsname}%
539 \expandafter\let\expandafter#1\csname bbl@normal@#2\endcsname

```

The next level of the code checks whether a user has defined a shorthand for himself with this character. First we check for a single character shorthand. If that doesn't exist we check for a shorthand with an argument.

```

540 \bbl@active@def#2\user@group{user@active}{language@active}%
541 \bbl@active@def#2\language@group{language@active}{system@active}%
542 \bbl@active@def#2\system@group{system@active}{normal@char}%

```

In order to do the right thing when a shorthand with an argument is used by itself at the end of the line we provide a definition for the case of an empty argument. For that case we let the shorthand character expand to its non-active self. Also, When a shorthand combination such as `'` ends up in a heading  $\TeX$  would see `\protect'\protect'`. To prevent this from happening a couple of shorthand needs to be defined at user level.

```

543 \expandafter\edef\csname\user@group @sh#2@@\endcsname
544     {\expandafter\noexpand\csname normal@char#2\endcsname}%
545 \expandafter\edef\csname\user@group @sh#2@\string\protect@\endcsname
546     {\expandafter\noexpand\csname user@active#2\endcsname}%

```

Finally, a couple of special cases are taken care of. (1) If we are making the right quote (`'`) active we need to change `\prim@s` as well. Also, make sure that a single `'` in math mode 'does the right thing'. (2) If we are using the caret (`^`) as a shorthand character special care should be taken to make sure math still works. Therefore an extra level of expansion is introduced with a check for math mode on the upper level.

```

547 \if\string'#2%
548     \let\prim@s\bbl@prim@s
549     \let\active@math@prime#1%
550 \fi}

```



Initiating a shorthand makes active the char. That is not strictly necessary but it is still done for backward compatibility. So we need to restore the original catcode at the end of package *and* and the end of the *ldf*.

```

551 \@ifpackagewith{babel}{KeepShorthandsActive}%
552 {\let\bbl@restoreactive\@gobble}%
553 {\def\bbl@restoreactive#1{%
554   \edef\bbl@tempa{%
555     \noexpand\AfterBabelLanguage\noexpand\CurrentOption
556     {\catcode'#1=\the\catcode'#1\relax}%
557     \noexpand\AtEndOfPackage{\catcode'#1=\the\catcode'#1\relax}}%
558   \bbl@tempa}%
559   \AtEndOfPackage{\let\bbl@restoreactive\@gobble}}
```

`\bbl@sh@select` This command helps the shorthand supporting macros to select how to proceed. Note that this macro needs to be expandable as do all the shorthand macros in order for them to work in expansion-only environments such as the argument of `\hyphenation`.

This macro expects the name of a group of shorthands in its first argument and a shorthand character in its second argument. It will expand to either `\bbl@firstcs` or `\bbl@scndcs`. Hence two more arguments need to follow it.

```

560 \def\bbl@sh@select#1#2{%
561   \expandafter\ifx\csname#1@sh@#2@sel\endcsname\relax
562     \bbl@afterelse\bbl@scndcs
563   \else
564     \bbl@afterfi\csname#1@sh@#2@sel\endcsname
565   \fi}
```

`\active@prefix` The command `\active@prefix` which is used in the expansion of active characters has a function similar to `\OT1-cmd` in that it `\protects` the active character whenever `\protect` is *not* `\@typeset@protect`.

```

566 \def\active@prefix#1{%
567   \ifx\protect\@typeset@protect
568   \else
```

When `\protect` is set to `\@unexpandable@protect` we make sure that the active character is *als not* expanded by inserting `\noexpand` in front of it. The `\@gobble` is needed to remove a token such as `\activechar:` (when the double colon was the active character to be dealt with).

```

569   \ifx\protect\@unexpandable@protect
570     \noexpand#1%
571   \else
572     \protect#1%
573   \fi
574   \expandafter\@gobble
575   \fi}
```

`\if@safe@actives` In some circumstances it is necessary to be able to change the expansion of an active character on the fly. For this purpose the switch `@safe@actives` is available. The setting of this switch should be checked in the first level expansion of `\active@char<char>`.

```

576 \newif\if@safe@actives
577 \@safe@activesfalse
```

`\bbl@restore@actives` When the output routine kicks in while the active characters were made “safe” this must be undone in the headers to prevent unexpected typeset results. For this situation we define a command to make them “unsafe” again.

```
578 \def\bbl@restore@actives{\if@safe@actives\@safe@activesfalse\fi}
```

`\bbl@activate` Both macros take one argument, like `\initiate@active@char`. The macro is used  
`\bbl@deactivate` to change the definition of an active character to expand to `\active@char⟨char⟩` in the case of `\bbl@activate`, or `\normal@char⟨char⟩` in the case of `\bbl@deactivate`.

```
579 \def\bbl@activate#1{%
580   \bbl@withactive{\expandafter\let\expandafter}#1%
581   \csname bbl@active@\string#1\endcsname}
582 \def\bbl@deactivate#1{%
583   \bbl@withactive{\expandafter\let\expandafter}#1%
584   \csname bbl@normal@\string#1\endcsname}
```

`\bbl@firstcs` These macros have two arguments. They use one of their arguments to build a  
`\bbl@scndcs` control sequence from.

```
585 \def\bbl@firstcs#1#2{\csname#1\endcsname}
586 \def\bbl@scndcs#1#2{\csname#2\endcsname}
```

`\declare@shorthand` The command `\declare@shorthand` is used to declare a shorthand on a certain level. It takes three arguments:

1. a name for the collection of shorthands, i.e. ‘system’, or ‘dutch’;
2. the character (sequence) that makes up the shorthand, i.e. `~` or `"a`;
3. the code to be executed when the shorthand is encountered.

```
587 \def\declare@shorthand#1#2{\@decl@short{#1}#2\@nil}
588 \def\@decl@short#1#2#3\@nil#4{%
589   \def\bbl@tempa{#3}%
590   \ifx\bbl@tempa\@empty
591     \expandafter\let\csname #1@sh@\string#2@sel\endcsname\bbl@scndcs
592     \@ifundefined{#1@sh@\string#2@}{}%
593     {\def\bbl@tempa{#4}%
594      \expandafter\ifx\csname#1@sh@\string#2@\endcsname\bbl@tempa
595      \else
596        \bbl@info
597        {Redefining #1 shorthand \string#2\\%
598         in language \CurrentOption}%
599        \fi}%
600     \@namedef{#1@sh@\string#2@}{#4}%
601   \else
602     \expandafter\let\csname #1@sh@\string#2@sel\endcsname\bbl@firstcs
603     \@ifundefined{#1@sh@\string#2@\string#3@}{}%
604     {\def\bbl@tempa{#4}%
605      \expandafter\ifx\csname#1@sh@\string#2@\string#3@\endcsname\bbl@tempa
606      \else
607        \bbl@info
608        {Redefining #1 shorthand \string#2\string#3\\%
609         in language \CurrentOption}%
610        \fi}%
611     \@namedef{#1@sh@\string#2@\string#3@}{#4}%
612   \fi}
```

`\textormath` Some of the shorthands that will be declared by the language definition files have to be usable in both text and mathmode. To achieve this the helper macro `\textormath` is provided.

```

613 \def\textormath{%
614   \ifmmode
615     \expandafter\@secondoftwo
616   \else
617     \expandafter\@firstoftwo
618   \fi}

```

`\user@group` The current concept of ‘shorthands’ supports three levels or groups of shorthands.  
`\language@group` For each level the name of the level or group is stored in a macro. The default is to  
`\system@group` have a user group; use language group ‘english’ and have a system group called ‘system’.

```

619 \def\user@group{user}
620 \def\language@group{english}
621 \def\system@group{system}

```

`\usesshorthands` This is the user level command to tell L<sup>A</sup>T<sub>E</sub>X that user level shorthands will be used in the document. It takes one argument, the character that starts a shorthand. First note that this is user level, and then initialize and activate the character for use as a shorthand character (ie, it’s active in the preamble). Languages can deactivate shorthands, so a starred version is also provided which activates them always after the language has been switched.

```

622 \def\usesshorthands{%
623   \@ifstar\bbl@usesesh@s{\bbl@usesesh@x{}}
624 \def\bbl@usesesh@s#1{%
625   \bbl@usesesh@x
626   {\AddBabelHook{babel-sh-\string#1}{afterextras}{\bbl@activate{#1}}}%
627   {#1}}
628 \def\bbl@usesesh@x#1#2{%
629   \bbl@ifshorthand{#2}%
630   {\def\user@group{user}%
631     \initiate@active@char{#2}%
632     #1%
633     \bbl@activate{#2}}%
634   {\bbl@error
635     {Cannot declare a shorthand turned off (\string#2)}
636     {Sorry, but you cannot use shorthands which have been\\%
637       turned off in the package options}}}

```

`\defineshorthand` Currently we only support two groups of user level shorthands, named internally `user` and `user@<lang>` (language-dependent user shorthands). By default, only the first one is taken into account, but if the former is also used (in the optional argument of `\defineshorthand`) a new level is inserted for it (`user@generic`, done by `\bbl@set@user@generic`); we make also sure `{}` and `\protect` are taken into account in this new top level.

```

638 \def\user@language@group{user@\language@group}
639 \def\bbl@set@user@generic#1#2{%
640   \@ifundefined{user@generic@active#1}%
641   {\bbl@active@def#1\user@language@group{user@active}{user@generic@active}%
642     \bbl@active@def#1\user@group{user@generic@active}{\language@active}%
643     \expandafter\edef\csname#2@sh@#1@\endcsname{%
644       \expandafter\noexpand\csname normal@char#1\endcsname}%

```

```

645 \expandafter\edef\csname#2@sh@#1@\string\protect\endcsname{%
646 \expandafter\noexpand\csname user@active#1\endcsname}}%
647 \@empty}
648 \newcommand\defineshorthand[3][user]{%
649 \edef\bb@tempa{\zap@space#1 \@empty}%
650 \bb@for\bb@tempb\bb@tempa{%
651 \if*\expandafter\@car\bb@tempb\@nil
652 \edef\bb@tempb{user@\expandafter\@gobble\bb@tempb}%
653 \@expandtwoargs
654 \bb@set@user@generic{\expandafter\string\@car#2\@nil}\bb@tempb
655 \fi
656 \declare@shorthand{\bb@tempb}{#2}{#3}}

```

`\languageshorthands` A user level command to change the language from which shorthands are used. Unfortunately, babel currently does not keep track of defined groups, and therefore there is no way to catch a possible change in casing.

```

657 \def\languageshorthands#1{\def\language@group{#1}}

```

`\aliasshorthand` First the new shorthand needs to be initialized,

```

658 \def\aliasshorthand#1#2{%
659 \bb@ifshorthand{#2}%
660 {\expandafter\ifx\csname active@char\string#2\endcsname\relax
661 \ifx\document\@notprerr
662 \@notshorthand{#2}%
663 \else
664 \initiate@active@char{#2}%

```

Then, we define the new shorthand in terms of the original one, but note with `\aliasshorthands{"}{/}` is `\active@prefix /active@char/`, so we still need to let the latest to `\active@char`.

```

665 \expandafter\let\csname active@char\string#2\expandafter\endcsname
666 \csname active@char\string#1\endcsname
667 \expandafter\let\csname normal@char\string#2\expandafter\endcsname
668 \csname normal@char\string#1\endcsname
669 \bb@activate{#2}%
670 \fi
671 \fi}%
672 {\bb@error
673 {Cannot declare a shorthand turned off (\string#2)}
674 {Sorry, but you cannot use shorthands which have been\\%
675 turned off in the package options}}}

```

`\@notshorthand`

```

676 \def\@notshorthand#1{%
677 \bb@error{%
678 The character '\string #1' should be made a shorthand character;\\%
679 add the command \string\usesshorthands\string{#1\string} to
680 the preamble.\\%
681 I will ignore your instruction}}

```

`\shorthandon` The first level definition of these macros just passes the argument on to  
`\shorthandoff` `\bb@switch@sh`, adding `\@nil` at the end to denote the end of the list of characters.

```

682 \newcommand*\shorthandon[1]{\bb@switch@sh\@ne#1\@nnil}
683 \DeclareRobustCommand*\shorthandoff{%
684 \@ifstar{\bb@shorthandoff\tw@}{\bb@shorthandoff\z@}}

```

```

685 \def\bbl@shorthandoff#1#2{\bbl@switch@sh#1#2\@nnil}

\bbl@switch@sh The macro \bbl@switch@sh takes the list of characters apart one by one and
subsequently switches the category code of the shorthand character according to
the first argument of \bbl@switch@sh.
But before any of this switching takes place we make sure that the character we
are dealing with is known as a shorthand character. If it is, a macro such as
\active@char" should exist.
Switching off and on is easy – we just set the category code to ‘other’ (12) and
\active. With the starred version, the original catcode and the original definition,
saved in @initiate@active@char, are restored.

686 \def\bbl@switch@sh#1#2{%
687   \ifx#2\@nnil\else
688     \@ifundefined{bbl@active@\string#2}%
689     {\bbl@error
690       {I cannot switch ‘\string#2’ on or off--not a shorthand}%
691       {This character is not a shorthand. Maybe you made\\%
692         a typing mistake? I will ignore your instruction}}%
693     {\ifcase#1%
694       \catcode'#212\relax
695       \or
696       \catcode'#2\active
697       \or
698       \csname bbl@oricat@\string#2\endcsname
699       \csname bbl@oridef@\string#2\endcsname
700       \fi}%
701   \bbl@afterfi\bbl@switch@sh#1%
702 \fi}

```

## 7.5 Conditional loading of shorthands

Note the value is that at the expansion time, eg, in the preamble shorhands are usually deactivated.

```

703 \def\babelshorthand{\active@prefix\babelshorthand\bbl@putsh}
704 \def\bbl@putsh#1{%
705   \@ifundefined{bbl@active@\string#1}%
706   {\bbl@putsh@i#1\@empty\@nnil}%
707   {\csname bbl@active@\string#1\endcsname}}
708 \def\bbl@putsh@i#1#2\@nnil{%
709   \csname\language @sh@\string#1@%
710   \ifx\@empty#2\else\string#2@\fi\endcsname}
711 \ifx\bbl@opt@shorthands\@nnil\else
712   \let\bbl@s@initiate@active@char\initiate@active@char
713   \def\initiate@active@char#1{%
714     \bbl@ifshorthand{#1}{\bbl@s@initiate@active@char{#1}}{}}
715   \let\bbl@s@switch@sh\bbl@switch@sh
716   \def\bbl@switch@sh#1#2{%
717     \ifx#2\@nnil\else
718       \bbl@afterfi
719       \bbl@ifshorthand{#2}{\bbl@s@switch@sh#1{#2}}{\bbl@switch@sh#1}%
720     \fi}
721   \let\bbl@s@activate\bbl@activate
722   \def\bbl@activate#1{%
723     \bbl@ifshorthand{#1}{\bbl@s@activate{#1}}{}}

```

```

724 \let\bb@deactivate\bb@deactivate
725 \def\bb@deactivate#1{%
726   \bb@ifshorthand{#1}{\bb@deactivate{#1}}{}}
727 \fi

```

\bb@prim@s One of the internal macros that are involved in substituting \prime for each right quote in mathmode is \prim@s. This checks if the next character is a right quote. When the right quote is active, the definition of this macro needs to be adapted to look also for an active right quote; the hat could be active, too.

```

728 \def\bb@prim@s{%
729   \prime\futurelet\@let@token\bb@pr@m@s}
730 \def\bb@if@primes#1#2{%
731   \ifx#1\@let@token
732     \expandafter\@firstoftwo
733   \else\ifx#2\@let@token
734     \bb@afterelse\expandafter\@firstoftwo
735   \else
736     \bb@afterfi\expandafter\@secondoftwo
737   \fi\fi}
738 \begingroup
739 \catcode'\^=7 \catcode'\*=\active \lccode'\*='\^
740 \catcode'\ '=12 \catcode'\ "=\active \lccode'\ "='\ '
741 \lowercase{%
742   \gdef\bb@pr@m@s{%
743     \bb@if@primes" '%
744     \pr@@@s
745     {\bb@if@primes*\^{\pr@@@t\egroup}}}
746 \endgroup

```

Usually the ~ is active and expands to \penalty\@M\\_. When it is written to the .aux file it is written expanded. To prevent that and to be able to use the character ~ as a start character for a shorthand, it is redefined here as a one character shorthand on system level. The system declaration is in most cases redundant (when ~ is still a non-break space), and in some cases is inconvenient (if ~ has been redefined); however, for backward compatibility it is maintained (some existing documents may rely on the babel value).

```

747 \initiate@active@char{~}
748 \declare@shorthand{system}{~}{\leavevmode\nobreak\ }
749 \bb@activate{~}

```

\OT1dqpos The position of the double quote character is different for the OT1 and T1 encodings. It will later be selected using the \f@encoding macro. Therefore we define two macros here to store the position of the character in these encodings.

```

750 \expandafter\def\csname OT1dqpos\endcsname{127}
751 \expandafter\def\csname T1dqpos\endcsname{4}

```

When the macro \f@encoding is undefined (as it is in plain T<sub>E</sub>X) we define it here to expand to OT1

```

752 \ifx\f@encoding\undefined
753   \def\f@encoding{OT1}
754 \fi

```

## 7.6 Language attributes

Language attributes provide a means to give the user control over which features of the language definition files he wants to enable.

`\languageattribute` The macro `\languageattribute` checks whether its arguments are valid and then activates the selected language attribute. First check whether the language is known, and then process each attribute in the list.

```

755 \newcommand\languageattribute[2]{%
756   \def\bbl@tempc{#1}%
757   \bbl@fixname\bbl@tempc
758   \bbl@iflanguage\bbl@tempc{%
759     \@for\bbl@attr:=#2\do{%

```

We want to make sure that each attribute is selected only once; therefore we store the already selected attributes in `\bbl@known@attrs`. When that control sequence is not yet defined this attribute is certainly not selected before.

```

760     \ifx\bbl@known@attrs\@undefined
761       \in@false
762     \else

```

Now we need to see if the attribute occurs in the list of already selected attributes.

```

763       \@expandtwoargs\in@{,\bbl@tempc-\bbl@attr,}{,\bbl@known@attrs,}%
764     \fi

```

When the attribute was in the list we issue a warning; this might not be the users intention.

```

765     \ifin@
766       \bbl@warning{%
767         You have more than once selected the attribute '\bbl@attr'\%
768         for language #1}%
769     \else

```

When we end up here the attribute is not selected before. So, we add it to the list of selected attributes and execute the associated  $\TeX$ -code.

```

770     \edef\bbl@tempa{%
771       \noexpand\bbl@add@list
772       \noexpand\bbl@known@attrs{\bbl@tempc-\bbl@attr}}%
773     \bbl@tempa
774     \edef\bbl@tempa{\bbl@tempc-\bbl@attr}%
775     \expandafter\bbl@ifknown@trib\expandafter{\bbl@tempa}\bbl@attributes%
776     {\csname\bbl@tempc @attr\bbl@attr\endcsname}%
777     {\@attrerr{\bbl@tempc}{\bbl@attr}}%
778   \fi}}

```

This command should only be used in the preamble of a document.

```

779 \@onlypreamble\languageattribute

```

The error text to be issued when an unknown attribute is selected.

```

780 \newcommand*{\@attrerr}[2]{%
781   \bbl@error
782   {The attribute #2 is unknown for language #1.}%
783   {Your command will be ignored, type <return> to proceed}}

```

`\bbl@declare@ttribute` This command adds the new language/attribute combination to the list of known attributes.

Then it defines a control sequence to be executed when the attribute is used in a document. The result of this should be that the macro `\extras...` for the current language is extended, otherwise the attribute will not work as its code is removed from memory at `\begin{document}`.

```

784 \def\bbl@declare@tribute#1#2#3{%
785   \@expandtwoargs\in@{,#2,}{,\BabelModifiers,}%

```

```

786 \ifin@
787 \AfterBabelLanguage{#1}{\languageattribute{#1}{#2}}%
788 \fi
789 \bbl@add@list\bbl@attributes{#1-#2}%
790 \expandafter\def\csname#1@attr@#2\endcsname{#3}}

```

`\bbl@ifattributeset` This internal macro has 4 arguments. It can be used to interpret T<sub>E</sub>X code based on whether a certain attribute was set. This command should appear inside the argument to `\AtBeginDocument` because the attributes are set in the document preamble, *after* babel is loaded.

The first argument is the language, the second argument the attribute being checked, and the third and fourth arguments are the true and false clauses.

```

791 \def\bbl@ifattributeset#1#2#3#4{%

```

First we need to find out if any attributes were set; if not we're done.

```

792 \ifx\bbl@known@attrs \@undefined
793 \in@false
794 \else

```

The we need to check the list of known attributes.

```

795 \@expandtwoargs\in@{,#1-#2,}{,\bbl@known@attrs,}%
796 \fi

```

When we're this far `\ifin@` has a value indicating if the attribute in question was set or not. Just to be safe the code to be executed is 'thrown over the `\fi`'.

```

797 \ifin@
798 \bbl@afterelse#3%
799 \else
800 \bbl@afterfi#4%
801 \fi
802 }

```

`\bbl@add@list` This internal macro adds its second argument to a comma separated list in its first argument. When the list is not defined yet (or empty), it will be initiated

```

803 \def\bbl@add@list#1#2{%
804 \ifx#1\@undefined
805 \def#1{#2}%
806 \else
807 \ifx#1\@empty
808 \def#1{#2}%
809 \else
810 \edef#1{#1,#2}%
811 \fi
812 \fi
813 }

```

`\bbl@ifknown@ttrib` An internal macro to check whether a given language/attribute is known. The macro takes 4 arguments, the language/attribute, the attribute list, the T<sub>E</sub>X-code to be executed when the attribute is known and the T<sub>E</sub>X-code to be executed otherwise.

```

814 \def\bbl@ifknown@ttrib#1#2{%

```

We first assume the attribute is unknown.

```

815 \let\bbl@tempa \@secondoftwo

```

Then we loop over the list of known attributes, trying to find a match.

```

816 \@for\bbl@tempb:=#2\do{%

```



```

817 \expandafter\in@\expandafter{\expandafter,\bbl@tempb,}{, #1,}%
818 \ifin@
    When a match is found the definition of \bbl@tempa is changed.
819 \let\bbl@tempa\@firstoftwo
820 \else
821 \fi}%
    Finally we execute \bbl@tempa.
822 \bbl@tempa
823 }

```

`\bbl@clear@ttribs` This macro removes all the attribute code from L<sup>A</sup>T<sub>E</sub>X's memory at `\begin{document}` time (if any is present).

```

824 \def\bbl@clear@ttribs{%
825 \ifx\bbl@attributes\undefined\else
826 \@for\bbl@tempa:=\bbl@attributes\do{%
827 \expandafter\bbl@clear@ttrib\bbl@tempa.
828 }%
829 \let\bbl@attributes\undefined
830 \fi
831 }
832 \def\bbl@clear@ttrib#1-#2.{%
833 \expandafter\let\csname#1@attr@#2\endcsname\undefined}
834 \AtBeginDocument{\bbl@clear@ttribs}

```

## 7.7 Support for saving macro definitions

To save the meaning of control sequences using `\babel@save`, we use temporary control sequences. To save hash table entries for these control sequences, we don't use the name of the control sequence to be saved to construct the temporary name. Instead we simply use the value of a counter, which is reset to zero each time we begin to save new values. This works well because we release the saved meanings before we begin to save a new set of control sequence meanings (see `\selectlanguage` and `\originalTeX`). Note undefined macros are not undefined any more when saved – they are `\relax`'ed.

`\babel@savecnt` The initialization of a new save cycle: reset the counter to zero.

`\babel@beginsave` 835 `\def\babel@beginsave{\babel@savecnt\z@}`

Before it's forgotten, allocate the counter and initialize all.

```

836 \newcount\babel@savecnt
837 \babel@beginsave

```

`\babel@save` The macro `\babel@save<csname>` saves the current meaning of the control sequence `<csname>` to `\originalTeX`<sup>18</sup>. To do this, we let the current meaning to a temporary control sequence, the restore commands are appended to `\originalTeX` and the counter is incremented.

```

838 \def\babel@save#1{%
839 \expandafter\let\csname babel@number\babel@savecnt\endcsname#1\relax
840 \begingroup
841 \toks@ \expandafter{\originalTeX\let#1=}
842 \edef\x{\endgroup
843 \def\noexpand\originalTeX{\the\toks@ \expandafter\noexpand

```

<sup>18</sup>`\originalTeX` has to be expandable, i. e. you shouldn't let it to `\relax`.

```

844         \csname babel@number\babel@savecnt\endcsname\relax}}%
845     \x
846     \advance\babel@savecnt\@ne}

```

`\babel@savevariable` The macro `\babel@savevariable<variable>` saves the value of the variable. `<variable>` can be anything allowed after the `\the` primitive.

```

847 \def\babel@savevariable#1{\begingroup
848     \toks@\expandafter{\originalTeX #1}%
849     \edef\x{\endgroup
850         \def\noexpand\originalTeX{\the\toks@ \the#1\relax}}%
851     \x}

```

`\bbl@frenchspacing` Some languages need to have `\frenchspacing` in effect. Others don't want that.  
`\bbl@nonfrenchspacing` The command `\bbl@frenchspacing` switches it on when it isn't already in effect and `\bbl@nonfrenchspacing` switches it off if necessary.

```

852 \def\bbl@frenchspacing{%
853     \ifnum\the\sfcode'\.=\@m
854         \let\bbl@nonfrenchspacing\relax
855     \else
856         \frenchspacing
857         \let\bbl@nonfrenchspacing\nonfrenchspacing
858     \fi}
859 \let\bbl@nonfrenchspacing\nonfrenchspacing

```

## 7.8 Support for extending macros

`\addto` For each language four control sequences have to be defined that control the language-specific definitions. To be able to add something to these macro once they have been defined the macro `\addto` is introduced. It takes two arguments, a `<control sequence>` and `TEX`-code to be added to the `<control sequence>`. If the `<control sequence>` has not been defined before it is defined now. The control sequence could also expand to `\relax`, in which case a circular definition results. The net result is a stack overflow. Otherwise the replacement text for the `<control sequence>` is expanded and stored in a token register, together with the `TEX`-code to be added. Finally the `<control sequence>` is redefined, using the contents of the token register.

```

860 \def\addto#1#2{%
861     \ifx#1\@undefined
862         \def#1{#2}%
863     \else
864         \ifx#1\relax
865             \def#1{#2}%
866         \else
867             {\toks@\expandafter{#1#2}%
868             \xdef#1{\the\toks@}}%
869     \fi
870 \fi}

```

## 7.9 Hyphens

`\babelhyphenation` This macro saves hyphenation exceptions. Two macros are used to store them: `\bbl@hyphenation@` for the global ones and `\bbl@hyphenation<lang>` for language ones. See `\bbl@patterns` above for further details. We make sure there is a space between words when multiple commands are used.

```

871 \@onlypreamble\babelhyphenation
872 \AtEndOfPackage{%
873   \newcommand\babelhyphenation[2][\@empty]{%
874     \ifx\bbbl@hyphenation@relax
875       \let\bbbl@hyphenation@ \@empty
876     \fi
877     \ifx\bbbl@hyphlist \@empty\else
878       \bbbl@warning{%
879         You must not intermingle \string\selectlanguage\space and\%
880         \string\babelhyphenation\space or some exception will not\%
881         be taken into account. Reported}%
882       \fi
883     \ifx \@empty#1%
884       \protected@edef\bbbl@hyphenation@{\bbbl@hyphenation@\space#2}%
885     \else
886       \edef\bbbl@tempb{\zap@space#1 \@empty}%
887       \bbbl@for\bbbl@tempa\bbbl@tempb{%
888         \bbbl@fixname\bbbl@tempa
889         \bbbl@iflanguage\bbbl@tempa{%
890           \bbbl@csarg\protected@edef{hyphenation@\bbbl@tempa}{%
891             \@ifundefined{bbbl@hyphenation@\bbbl@tempa}%
892             \@empty
893             {\csname bbl@hyphenation@\bbbl@tempa\endcsname\space}%
894             #2}}}%
895     \fi}}

```

`\bbbl@allowhyphens` This macro makes hyphenation possible. Basically its definition is nothing more than `\nobreak \hskip 0pt plus 0pt`<sup>19</sup>.

```

896 \def\bbbl@allowhyphens{\nobreak\hskip\z@skip}
897 \def\bbbl@t@one{T1}
898 \def\allowhyphens{\ifx\cf@encoding\bbbl@t@one\else\bbbl@allowhyphens\fi}

```

`\babelhyphen` Macros to insert common hyphens. Note the space before @ in `\babelhyphen`. Instead of protecting it with `\DeclareRobustCommand`, which could insert a `\relax`, we use the same procedure as shorthands.

```

899 \newcommand\babelnullhyphen{\char\hyphenchar\font}
900 \def\babelhyphen{\active@prefix\babelhyphen\bbbl@hyphen}
901 \def\bbbl@hyphen{%
902   \@ifstar{\bbbl@hyphen@i @}{\bbbl@hyphen@i \@empty}}
903 \def\bbbl@hyphen@i#1#2{%
904   \@ifundefined{bbbl@hy@#1#2 \@empty}%
905   {\csname bbl@#1usehyphen\endcsname{\discretionary{#2}{\{#2}}}%
906   {\csname bbl@hy@#1#2 \@empty\endcsname}}

```

The following two commands are used to wrap the “hyphen” and set the behaviour of the rest of the word – the version with a single @ is used when further hyphenation is allowed, while that with @@ if no more hyphen are allowed. In both cases, if the hyphen is preceded by a positive space, breaking after the hyphen is disallowed.

There should not be a discretionary after a hyphen at the beginning of a word, so it is prevented if preceded by a skip. Unfortunately, this does handle cases like “(-suffix)”. `\nobreak` is always preceded by `\leavevmode`, in case the shorthand starts a paragraph.

<sup>19</sup> $\TeX$  begins and ends a word for hyphenation at a glue node. The penalty prevents a linebreak at this glue node.

```

907 \def\bb@usehyphen#1{%
908   \leavevmode
909   \ifdim\lastskip>\z@\mbox{#1}\nobreak\else\nobreak#1\fi
910   \hskip\z@skip}
911 \def\bb@usehyphen#1{%
912   \leavevmode\ifdim\lastskip>\z@\mbox{#1}\else#1\fi}

```

The following macro inserts the hyphen char.

```

913 \def\bb@hyphenchar{%
914   \ifnum\hyphenchar\font=\m@ne
915     \babe\nullhyphen
916   \else
917     \char\hyphenchar\font
918   \fi}

```

Finally, we define the hyphen “types”. Their names will not change, so you may use them in ldf’s.

```

919 \def\bb@hy@soft{\bb@usehyphen{\discretionary{\bb@hyphenchar}{}}{}}
920 \def\bb@hy@soft{\bb@usehyphen{\discretionary{\bb@hyphenchar}{}}{}}
921 \def\bb@hy@hard{\bb@usehyphen\bb@hyphenchar}
922 \def\bb@hy@hard{\bb@usehyphen\bb@hyphenchar}
923 \def\bb@hy@nobreak{\bb@usehyphen{\mbox{\bb@hyphenchar}\nobreak}}
924 \def\bb@hy@nobreak{\mbox{\bb@hyphenchar}}
925 \def\bb@hy@repeat{%
926   \bb@usehyphen{%
927     \discretionary{\bb@hyphenchar}{\bb@hyphenchar}{\bb@hyphenchar}%
928     \nobreak}}
929 \def\bb@hy@repeat{%
930   \bb@usehyphen{%
931     \discretionary{\bb@hyphenchar}{\bb@hyphenchar}{\bb@hyphenchar}}}
932 \def\bb@hy@empty{\hskip\z@skip}
933 \def\bb@hy@empty{\discretionary{}{}{}}

```

`\bb@disc` For some languages the macro `\bb@disc` is used to ease the insertion of discretionaries for letters that behave ‘abnormally’ at a breakpoint.

```

934 \def\bb@disc#1#2{\nobreak\discretionary{#2-}{}{#1}\bb@allowhyphens}

```

## 7.10 Macros common to a number of languages

`\set@low@box` The following macro is used to lower quotes to the same level as the comma. It prepares its argument in box register 0.

```

935 \def\set@low@box#1{\setbox\tw@\hbox{,}\setbox\z@\hbox{#1}%
936   \dimen\z@\ht\z@ \advance\dimen\z@ -\ht\tw@%
937   \setbox\z@\hbox{\lower\dimen\z@ \box\z@}\ht\z@\ht\tw@ \dp\z@\dp\tw@}

```

`\save@sf@q` The macro `\save@sf@q` is used to save and reset the current space factor.

```

938 \def\save@sf@q#1{\leavevmode
939   \begingroup
940   \edef\@SF{\spacefactor\the\spacefactor}#1\@SF
941   \endgroup}

```

## 7.11 Making glyphs available

The file `babel.dtx`<sup>20</sup> makes a number of glyphs available that either do not exist in the OT1 encoding and have to be ‘faked’, or that are not accessible through

<sup>20</sup>The file described in this section has version number v3.9g, and was last revised on 2013/10/10.

Tlenc.def.

## 7.12 Quotation marks

`\quotedblbase` In the T1 encoding the opening double quote at the baseline is available as a separate character, accessible via `\quotedblbase`. In the OT1 encoding it is not available, therefore we make it available by lowering the normal open quote character to the baseline.

```
942 \ProvideTextCommand{\quotedblbase}{OT1}{%
943   \save@sf@q{\set@low@box{\textquotedblright\}%
944     \box\z@\kern-.04em\bbl@allowhyphens}}
```

Make sure that when an encoding other than OT1 or T1 is used this glyph can still be typeset.

```
945 \ProvideTextCommandDefault{\quotedblbase}{%
946   \UseTextSymbol{OT1}{\quotedblbase}}
```

`\quotesinglbase` We also need the single quote character at the baseline.

```
947 \ProvideTextCommand{\quotesinglbase}{OT1}{%
948   \save@sf@q{\set@low@box{\textquoteright\}%
949     \box\z@\kern-.04em\bbl@allowhyphens}}
```

Make sure that when an encoding other than OT1 or T1 is used this glyph can still be typeset.

```
950 \ProvideTextCommandDefault{\quotesinglbase}{%
951   \UseTextSymbol{OT1}{\quotesinglbase}}
```

`\guillemotleft` The guillemet characters are not available in OT1 encoding. They are faked.

```
\guillemotright 952 \ProvideTextCommand{\guillemotleft}{OT1}{%
953   \ifmmode
954     \ll
955   \else
956     \save@sf@q{\nobreak
957       \raise.2ex\hbox{$\scriptscriptstyle\ll$}\bbl@allowhyphens}%
958   \fi}
959 \ProvideTextCommand{\guillemotright}{OT1}{%
960   \ifmmode
961     \gg
962   \else
963     \save@sf@q{\nobreak
964       \raise.2ex\hbox{$\scriptscriptstyle\gg$}\bbl@allowhyphens}%
965   \fi}
```

Make sure that when an encoding other than OT1 or T1 is used these glyphs can still be typeset.

```
966 \ProvideTextCommandDefault{\guillemotleft}{%
967   \UseTextSymbol{OT1}{\guillemotleft}}
968 \ProvideTextCommandDefault{\guillemotright}{%
969   \UseTextSymbol{OT1}{\guillemotright}}
```

`\guilsinglleft` The single guillemets are not available in OT1 encoding. They are faked.

```
\guilsinglright 970 \ProvideTextCommand{\guilsinglleft}{OT1}{%
971   \ifmmode
972     <%
973   \else
974     \save@sf@q{\nobreak
```

```

975 \raise.2ex\hbox{$\scriptscriptstyle<$}\bbl@allowhyphens}%
976 \fi}
977 \ProvideTextCommand{\guilsinglright}{OT1}{%
978 \ifmmode
979 >%
980 \else
981 \save@sf@q{\nobreak
982 \raise.2ex\hbox{$\scriptscriptstyle>$}\bbl@allowhyphens}%
983 \fi}

```

Make sure that when an encoding other than OT1 or T1 is used these glyphs can still be typeset.

```

984 \ProvideTextCommandDefault{\guilsinglleft}{%
985 \UseTextSymbol{OT1}{\guilsinglleft}}
986 \ProvideTextCommandDefault{\guilsinglright}{%
987 \UseTextSymbol{OT1}{\guilsinglright}}

```

## 7.13 Letters

`\ij` The dutch language uses the letter ‘ij’. It is available in T1 encoded fonts, but not  
`\IJ` in the OT1 encoded fonts. Therefore we fake it for the OT1 encoding.

```

988 \DeclareTextCommand{\ij}{OT1}{%
989 i\kern-0.02em\bbl@allowhyphens j}
990 \DeclareTextCommand{\IJ}{OT1}{%
991 I\kern-0.02em\bbl@allowhyphens J}
992 \DeclareTextCommand{\ij}{T1}{\char188}
993 \DeclareTextCommand{\IJ}{T1}{\char156}

```

Make sure that when an encoding other than OT1 or T1 is used these glyphs can still be typeset.

```

994 \ProvideTextCommandDefault{\ij}{%
995 \UseTextSymbol{OT1}{\ij}}
996 \ProvideTextCommandDefault{\IJ}{%
997 \UseTextSymbol{OT1}{\IJ}}

```

`\dj` The croatian language needs the letters `\dj` and `\DJ`; they are available in the T1  
`\DJ` encoding, but not in the OT1 encoding by default.

Some code to construct these glyphs for the OT1 encoding was made available to me by Stipcevic Mario, (stipcevic@olimp.irb.hr).

```

998 \def\crrtic@{\hrule height0.1ex width0.3em}
999 \def\crttic@{\hrule height0.1ex width0.33em}
1000 \def\ddj@{%
1001 \setbox0\hbox{d}\dimen@=\ht0
1002 \advance\dimen@lex
1003 \dimen@.45\dimen@
1004 \dimen@ii\expandafter\rem@pt\the\fontdimen\@ne\font\dimen@
1005 \advance\dimen@ii.5ex
1006 \leavevmode\rlap{\raise\dimen@\hbox{\kern\dimen@ii\vbox{\crrtic@}}}}
1007 \def\DDJ@{%
1008 \setbox0\hbox{D}\dimen@=.55\ht0
1009 \dimen@ii\expandafter\rem@pt\the\fontdimen\@ne\font\dimen@
1010 \advance\dimen@ii.15ex % correction for the dash position
1011 \advance\dimen@ii-.15\fontdimen7\font % correction for cmtt font
1012 \dimen\thr@@\expandafter\rem@pt\the\fontdimen7\font\dimen@
1013 \leavevmode\rlap{\raise\dimen@\hbox{\kern\dimen@ii\vbox{\crttic@}}}}
1014 %

```

```

1015 \DeclareTextCommand{\dj}{OT1}{\ddj@ d}
1016 \DeclareTextCommand{\DJ}{OT1}{\DDJ@ D}

```

Make sure that when an encoding other than OT1 or T1 is used these glyphs can still be typeset.

```

1017 \ProvideTextCommandDefault{\dj}{%
1018   \UseTextSymbol{OT1}{\dj}}
1019 \ProvideTextCommandDefault{\DJ}{%
1020   \UseTextSymbol{OT1}{\DJ}}

```

`\SS` For the T1 encoding `\SS` is defined and selects a specific glyph from the font, but for other encodings it is not available. Therefore we make it available here.

```

1021 \DeclareTextCommand{\SS}{OT1}{SS}
1022 \ProvideTextCommandDefault{\SS}{\UseTextSymbol{OT1}{\SS}}

```

## 7.14 Shorthands for quotation marks

Shorthands are provided for a number of different quotation marks, which make them usable both outside and inside mathmode.

`\glq` The ‘german’ single quotes.

```

\grq 1023 \ProvideTextCommand{\glq}{OT1}{%
1024   \textormath{\quotesinglbase}{\mbox{\quotesinglbase}}}
1025 \ProvideTextCommand{\glq}{T1}{%
1026   \textormath{\quotesinglbase}{\mbox{\quotesinglbase}}}
1027 \ProvideTextCommandDefault{\glq}{\UseTextSymbol{OT1}\glq}

```

The definition of `\grq` depends on the fontencoding. With T1 encoding no extra kerning is needed.

```

1028 \ProvideTextCommand{\grq}{T1}{%
1029   \textormath{\textquoteleft}{\mbox{\textquoteleft}}}
1030 \ProvideTextCommand{\grq}{OT1}{%
1031   \save@sf@q{\kern-.0125em%
1032     \textormath{\textquoteleft}{\mbox{\textquoteleft}}}%
1033     \kern.07em\relax}}
1034 \ProvideTextCommandDefault{\grq}{\UseTextSymbol{OT1}\grq}

```

`\glqq` The ‘german’ double quotes.

```

\grqq 1035 \ProvideTextCommand{\glqq}{OT1}{%
1036   \textormath{\quotedblbase}{\mbox{\quotedblbase}}}
1037 \ProvideTextCommand{\glqq}{T1}{%
1038   \textormath{\quotedblbase}{\mbox{\quotedblbase}}}
1039 \ProvideTextCommandDefault{\glqq}{\UseTextSymbol{OT1}\glqq}

```

The definition of `\grqq` depends on the fontencoding. With T1 encoding no extra kerning is needed.

```

1040 \ProvideTextCommand{\grqq}{T1}{%
1041   \textormath{\textquotedblleft}{\mbox{\textquotedblleft}}}
1042 \ProvideTextCommand{\grqq}{OT1}{%
1043   \save@sf@q{\kern-.07em%
1044     \textormath{\textquotedblleft}{\mbox{\textquotedblleft}}}%
1045     \kern.07em\relax}}
1046 \ProvideTextCommandDefault{\grqq}{\UseTextSymbol{OT1}\grqq}

```

`\flq` The ‘french’ single guillemets.

```

\frq 1047 \ProvideTextCommand{\flq}{OT1}{%

```

```

1048 \textormath{\guilsinglleft}{\mbox{\guilsinglleft}}}
1049 \ProvideTextCommand{\flq}{T1}{%
1050 \textormath{\guilsinglleft}{\mbox{\guilsinglleft}}}
1051 \ProvideTextCommandDefault{\flq}{\UseTextSymbol{OT1}\flq}

1052 \ProvideTextCommand{\frq}{OT1}{%
1053 \textormath{\guilsinglright}{\mbox{\guilsinglright}}}
1054 \ProvideTextCommand{\frq}{T1}{%
1055 \textormath{\guilsinglright}{\mbox{\guilsinglright}}}
1056 \ProvideTextCommandDefault{\frq}{\UseTextSymbol{OT1}\frq}

```

`\flqq` The ‘french’ double guillemets.

```

\frqq 1057 \ProvideTextCommand{\flqq}{OT1}{%
1058 \textormath{\guillemotleft}{\mbox{\guillemotleft}}}
1059 \ProvideTextCommand{\flqq}{T1}{%
1060 \textormath{\guillemotleft}{\mbox{\guillemotleft}}}
1061 \ProvideTextCommandDefault{\flqq}{\UseTextSymbol{OT1}\flqq}

1062 \ProvideTextCommand{\frqq}{OT1}{%
1063 \textormath{\guillemotright}{\mbox{\guillemotright}}}
1064 \ProvideTextCommand{\frqq}{T1}{%
1065 \textormath{\guillemotright}{\mbox{\guillemotright}}}
1066 \ProvideTextCommandDefault{\frqq}{\UseTextSymbol{OT1}\frqq}

```

## 7.15 Umlauts and trema’s

The command `\` needs to have a different effect for different languages. For German for instance, the ‘umlaut’ should be positioned lower than the default position for placing it over the letters a, o, u, A, O and U. When placed over an e, i, E or I it can retain its normal position. For Dutch the same glyph is always placed in the lower position.

`\umlauthigh` To be able to provide both positions of `\` we provide two commands to switch the  
`\umlautlow` positioning, the default will be `\umlauthigh` (the normal positioning).

```

1067 \def\umlauthigh{%
1068 \def\bb@umlauta##1{\leavevmode\bgroup%
1069 \expandafter\accent\csname f@encoding dqpos\endcsname
1070 ##1\bb@allowhyphens\egroup}%
1071 \let\bb@umlaute\bb@umlauta}
1072 \def\umlautlow{%
1073 \def\bb@umlauta{\protect\lower@umlaut}}
1074 \def\umlautelower{%
1075 \def\bb@umlaute{\protect\lower@umlaut}}
1076 \umlauthigh

```

`\lower@umlaut` The command `\lower@umlaut` is used to position the `\` closer to the letter. We want the umlaut character lowered, nearer to the letter. To do this we need an extra *⟨dimen⟩* register.

```

1077 \expandafter\ifx\csname U@D\endcsname\relax
1078 \csname newdimen\endcsname U@D
1079 \fi

```

The following code fools  $\text{T}_{\text{E}}\text{X}$ ’s `make\_accent` procedure about the current x-height of the font to force another placement of the umlaut character. First we have to save the current x-height of the font, because we’ll change this font dimension and this is always done globally.



Then we compute the new x-height in such a way that the umlaut character is lowered to the base character. The value of .45ex depends on the METAFONT parameters with which the fonts were built. (Just try out, which value will look best.) If the new x-height is too low, it is not changed. Finally we call the `\accent` primitive, reset the old x-height and insert the base character in the argument.

```

1080 \def\lower@umlaut#1{%
1081   \leavevmode\bggroup
1082     \U@D lex%
1083     {\setbox\z@\hbox{%
1084       \expandafter\char\csname\f@encoding dqpos\endcsname}%
1085       \dimen@ -.45ex\advance\dimen@\ht\z@
1086       \ifdim lex<\dimen@ \fontdimen5\font\dimen@ \fi}%
1087     \expandafter\accent\csname\f@encoding dqpos\endcsname
1088     \fontdimen5\font\U@D #1%
1089   \egroup}

```

For all vowels we declare `\` to be a composite command which uses `\bbl@umlauta` or `\bbl@umlaute` to position the umlaut character. We need to be sure that these definitions override the ones that are provided when the package `fontenc` with option `OT1` is used. Therefore these declarations are postponed until the beginning of the document. Note these definitions only apply to some languages, but `babel` sets them for *all* languages – you may want to redefine `\bbl@umlauta` and/or `\bbl@umlaute` for a language in the corresponding `ldf` (using the `babel` switching mechanism, of course).

```

1090 \AtBeginDocument{%
1091   \DeclareTextCompositeCommand{"}{OT1}{a}{\bbl@umlauta{a}}%
1092   \DeclareTextCompositeCommand{"}{OT1}{e}{\bbl@umlaute{e}}%
1093   \DeclareTextCompositeCommand{"}{OT1}{i}{\bbl@umlaute{i}}%
1094   \DeclareTextCompositeCommand{"}{OT1}{\i}{\bbl@umlaute{i}}%
1095   \DeclareTextCompositeCommand{"}{OT1}{o}{\bbl@umlauta{o}}%
1096   \DeclareTextCompositeCommand{"}{OT1}{u}{\bbl@umlauta{u}}%
1097   \DeclareTextCompositeCommand{"}{OT1}{A}{\bbl@umlauta{A}}%
1098   \DeclareTextCompositeCommand{"}{OT1}{E}{\bbl@umlaute{E}}%
1099   \DeclareTextCompositeCommand{"}{OT1}{I}{\bbl@umlaute{I}}%
1100   \DeclareTextCompositeCommand{"}{OT1}{O}{\bbl@umlauta{O}}%
1101   \DeclareTextCompositeCommand{"}{OT1}{U}{\bbl@umlauta{U}}%
1102 }

```

## 7.16 Multiencoding strings

The aim following commands is to provide a common interface for strings in several encodings. They also contains several hooks which can be used by `luatex` and `xetex`. The code is organized here with pseudo-guards, so we start with the basic commands.

The command `\SetCase` provides a way to change the behaviour of `\MakeUppercase` and `\MakeLowercase`. We need to patch `\@uclclist`, but it is done once and only if `\SetCase` is used.

```

1103 \def\bbl@patchuclc{%
1104   \let\bbl@patchuclc\relax
1105   \@ifundefined{bbl@tolower}{%
1106     \g@addto@macro\@uclclist{%
1107       \reserved@a{\reserved@a@gobble}% stops processing the list
1108       \@ifundefined{\language @bbl@uclc}% and resumes it
1109       {\reserved@a}%

```

```

1110      {\csname\language @bbl@ucl\endcsname}%
1111      {\bbl@tolower\empty}\bbl@toupper\empty}}%
1112      \gdef\bbl@tolower{\csname\language @bbl@lc\endcsname}%
1113      \gdef\bbl@toupper{\csname\language @bbl@uc\endcsname}}}%

```

The command to be used in the ldf.

```

1114 <<*Define SetCase>> ≡
1115 \newcommand\SetCase[3][]{%
1116   \bbl@patchucl
1117   \bbl@forlang\bbl@tempa{%
1118     \expandafter\bbl@stringdef
1119     \csname\bbl@tempa @bbl@ucl\endcsname{\reserved@a##1}%
1120     \expandafter\bbl@stringdef
1121     \csname\bbl@tempa @bbl@uc\endcsname{##2}%
1122     \expandafter\bbl@stringdef
1123     \csname\bbl@tempa @bbl@lc\endcsname{##3}}}%
1124 <</Define SetCase>>

```

Now we define \SetStringLoop. The current definition is somewhat complicated because we need a count, but \count@ and the like are not under our control (remember \SetString may call hooks).

```

1125 <<*Define SetStringLoop>> ≡
1126 \def\SetStringLoop{\afterassignment\bbl@sclp\def\bbl@templ####1}%
1127 \def\bbl@sclp##1{%
1128   \def\bbl@tempk{\@ne}%
1129   \@for\bbl@tempm:=##1\do{%
1130     \count@\bbl@tempk\relax % Because of stringprocess event
1131     \toks@\expandafter{\bbl@tempm}%
1132     \edef\bbl@tempn{%
1133       \expandafter\noexpand
1134       \csname\bbl@templ{\romannumeral\count@}\endcsname{\the\toks@}}%
1135     \advance\count@\@ne
1136     \edef\bbl@tempk{\the\count@}%
1137     \expandafter\SetString\bbl@tempn}}%
1138 <</Define SetStringLoop>>

```

This is the main command. With the first use it is redefined to omit the basic setup in subsequent blocks. We make sure strings contain actual letters in the range 128-255, not active characters.

```

1139 \def\bbl@aftercmds#1{%
1140   \toks@\expandafter{\bbl@scafter#1}%
1141   \xdef\bbl@scafter{\the\toks@}}
1142 \def\bbl@tglobal#1{\global\let#1#1}
1143 \def\bbl@recatcode#1{%
1144   \@tempcnta="7F
1145   \def\bbl@tempa{%
1146     \ifnum\@tempcnta>"FF\else
1147       \catcode\@tempcnta=#1\relax
1148       \advance\@tempcnta\@ne
1149       \expandafter\bbl@tempa
1150     \fi}%
1151   \bbl@tempa}
1152 \onlypreamble\StartBabelCommands
1153 \def\StartBabelCommands{%
1154   \begingroup
1155   \bbl@recatcode{11}%

```

```

1156 <<Define SetStringLoop>>
1157 <<Define SetCase>>
1158 \def\bbbl@provstring##1##2{%
1159   \providecommand##1{##2}%
1160   \bbbl@tglobal##1}%
1161 \global\let\bbbl@scafter\@empty
1162 \let\StartBabelCommands\bbbl@startcmds
1163 \ifx\BabelLanguages\relax
1164   \let\BabelLanguages\CurrentOption
1165 \fi
1166 \begingroup
1167 \let\bbbl@screset\@nnil % local flag - disable 1st stopcommands
1168 \StartBabelCommands}
1169 \def\bbbl@startcmds{%
1170   \ifx\bbbl@screset\@nnil\else
1171     \bbbl@usehooks{stopcommands}{}%
1172   \fi
1173 \endgroup
1174 \begingroup
1175 \@ifstar
1176   {\ifx\bbbl@opt@strings\@nnil
1177     \let\bbbl@opt@strings\BabelStringsDefault
1178     \fi
1179     \bbbl@startcmds@i}%
1180   \bbbl@startcmds@i}
1181 \def\bbbl@startcmds@i#1#2{%
1182   \edef\bbbl@L{\zap@space#1 \@empty}%
1183   \edef\bbbl@G{\zap@space#2 \@empty}%
1184   \bbbl@startcmds@ii}

```

Parse the encoding info to get the label, input, and font parts.

Select the behaviour of \SetString. There are two main cases, depending of if there is an optional argument: without it and strings=encoded, strings are defined always; otherwise, they are set only if they are still undefined (ie, fallback values). With labelled blocks and strings=encoded, define the strings, but with another value, define strings only if the current label or font encoding is the value of strings; otherwise (ie, no strings or a block whose label is not in strings=) do nothing.

We presume the current block is not loaded, and therefore set (above) a couple of default values to gobble the arguments. Then, these macros are redefined if necessary according to several parameters.

```

1185 \newcommand\bbbl@startcmds@ii[1][\@empty]{%
1186   \let\SetString\@gobbletwo
1187   \let\bbbl@stringdef\@gobbletwo
1188   \let\AfterBabelCommands\@gobble
1189   \ifx\@empty#1%
1190     \def\bbbl@sc@label{generic}%
1191     \def\bbbl@encstring##1##2{%
1192       \ProvideTextCommandDefault##1{##2}%
1193       \bbbl@tglobal##1%
1194       \expandafter\bbbl@tglobal\csname\string?\string##1\endcsname}%
1195     \let\bbbl@sctest\in@true
1196   \else
1197     \let\bbbl@sc@charset\space % <- zapped below
1198     \let\bbbl@sc@fontenc\space % <- " "

```

```

1199 \def\bbl@tempa##1=##2\@nil{%
1200   \bbl@csarg\edef{sc@\zap@space##1 \@empty}{##2 }}%
1201 \bbl@for\bbl@tempb{label=#1}{\expandafter\bbl@tempa\bbl@tempb\@nil}%
1202 \def\bbl@tempa##1 ##2{% space -> comma
1203   ##1%
1204   \ifx\@empty##2\else\ifx,##1,\else,\fi\bbl@afterfi\bbl@tempa##2\fi}%
1205 \edef\bbl@sc@fontenc{\expandafter\bbl@tempa\bbl@sc@fontenc\@empty}%
1206 \edef\bbl@sc@label{\expandafter\zap@space\bbl@sc@label\@empty}%
1207 \edef\bbl@sc@charset{\expandafter\zap@space\bbl@sc@charset\@empty}%
1208 \def\bbl@encstring##1##2{%
1209   \bbl@for\bbl@tempc\bbl@sc@fontenc{%
1210     \@ifundefined{T@\bbl@tempc}%
1211     {}%
1212     {ProvideTextCommand##1\bbl@tempc{##2}%
1213       \bbl@tglobal##1%
1214       \expandafter
1215       \bbl@tglobal\csname\bbl@tempc\string##1\endcsname}}}%
1216 \def\bbl@sctest{%
1217   \@expandtwoargs
1218   \in@{,\bbl@opt@strings,},{,\bbl@sc@label,\bbl@sc@fontenc,}}%
1219 \fi
1220 \bbl@scswitch{%
1221   \ifx\bbl@opt@strings\@nnil % ie, no strings
1222     \let\bbl@stringdef\bbl@encstring % for \SetCase
1223   \else\ifx\bbl@opt@strings\relax % ie, strings with no value
1224     \let\AfterBabelCommands\bbl@aftercmds
1225     \let\SetString\bbl@setstring
1226     \let\bbl@stringdef\bbl@encstring
1227   \else % ie, strings=value
1228     \bbl@sctest
1229     \ifin@
1230       \let\AfterBabelCommands\bbl@aftercmds
1231       \let\SetString\bbl@setstring
1232       \let\bbl@stringdef\bbl@provstring
1233     \fi\fi\fi}%
1234 \ifx\@empty#1%
1235   \@expandtwoargs
1236   \bbl@usehooks{defaultcommands}{}%
1237 \else
1238   \@expandtwoargs
1239   \bbl@usehooks{encodedcommands}{\bbl@sc@charset}{\bbl@sc@fontenc}}%
1240 \fi}

```

There are two versions of `\bbl@scswitch`. The first version is used when `ldfs` are read, and with it blocks not intended for the current language (as set in `\CurrentOption`) are ignored; it also makes sure `\langle group \rangle \langle language \rangle` is reset, but only once (`\bbl@screset` is used to keep track of this). The second version is used in the preamble and packages loaded after `babel` and defines strings for all known languages. The macro `\bbl@forlang` loops `\bbl@L` but its body is executed only if the value is in `\BabelLanguages` (inside `babel`) or `\date \langle language \rangle` is defined (after `babel` has been loaded).

```

1241 \def\bbl@forlang#1#2{%
1242   \bbl@for#1\bbl@L{%
1243     \@expandtwoargs\in@{,#1,},{,\BabelLanguages,}%
1244     \ifin#2\fi}}
1245 \def\bbl@scswitch#1{%

```

```

1246 #1\relax
1247 \bbl@scswitch@i
1248 \bbl@forlang\bbl@tempa{%
1249   \ifx\bbl@G\@empty\else
1250     \ifx\SetString\@gobbletwo\else
1251       \edef\bbl@GL{\bbl@G\bbl@tempa}%
1252       \@expandtwoargs\in@{,\bbl@GL,}{,\bbl@screset,}%
1253       \ifin@\else
1254         \global\expandafter\let\csname\bbl@GL\endcsname\@undefined
1255         \xdef\bbl@screset{\bbl@screset,\bbl@GL}%
1256       \fi
1257     \fi
1258   \fi}}
1259 \def\bbl@scswitch@i{%
1260   \ifx\bbl@G\@empty
1261     \def\SetString##1##2{%
1262       \bbl@error{Missing group for string \string##1}%
1263       {You must assign strings to some category, typically\\%
1264         captions or extras, but you set none}}%
1265     \fi}
1266 \AtEndOfPackage{%
1267   \def\bbl@scswitch#1{#1\relax\bbl@scswitch@i}%
1268   \def\bbl@forlang#1#2{\bbl@for#1\bbl@L{\@ifundefined{date#1}{}{#2}}}}
1269 \onlypreamble\EndBabelCommands
1270 \def\EndBabelCommands{%
1271   \bbl@usehooks{stopcommands}}}%
1272 \endgroup
1273 \endgroup
1274 \bbl@scafter}

```

First save the “switcher”. Create it if undefined. Strings are defined only if undefined (ie, like \providescommand). With the event stringprocess you can preprocess the string by manipulating the value of \BabelString. If there are several hooks assigned to this event, preprocessing is done in the same order as defined. Finally, the string is set.

```

1275 \def\bbl@scset#1#2{\def#1{#2}}
1276 \def\bbl@setstring#1#2{%
1277   \bbl@forlang\bbl@tempa{%
1278     \edef\bbl@LC{\bbl@tempa\expandafter\@gobble\string#1}%
1279     \@ifundefined{\bbl@LC}% eg, \germanchaptername
1280     {\global\expandafter
1281       \bbl@add\csname\bbl@G\bbl@tempa\expandafter\endcsname\expandafter
1282       {\expandafter\bbl@scset\expandafter#1\csname\bbl@LC\endcsname}}%
1283     }%
1284   \def\BabelString{#2}%
1285   \bbl@usehooks{stringprocess}}}%
1286 \expandafter\bbl@stringdef
1287   \csname\bbl@LC\expandafter\endcsname\expandafter{\BabelString}}

```

Macros to deal with case mapping for hyphenation. To decide if the document is monolingual or multilingual, we make a rough guess – just see if there is a comma in the languages list, built in the first pass of the package options.

```

1288 \def\SetHyphenMap#1{%
1289   \bbl@forlang\bbl@tempa{%
1290     \expandafter\bbl@stringdef
1291     \csname\bbl@tempa @bbl@hyphenmap\endcsname{#1}}

```

```

1292 \AtEndOfPackage{%
1293   \ifx\bbl@hmapopt\undefined
1294     \@expandtwoargs\in{,}{\bbl@language@opts}%
1295     \chardef\bbl@hmapopt\ifin@4\else\@ne\fi
1296   \fi}
1297 % \end{macrocode}
1298 %
1299 %   There are 3 helper macros which do most of the work for you.
1300 %
1301 % \begin{macrocode}
1302 \newcommand\BabelLower[2]{% one to one.
1303   \ifnum\lccode#1=#2\else
1304     \babel@savevariable{\lccode#1}%
1305     \lccode#1=#2\relax
1306   \fi}
1307 \newcommand\BabelLowerMM[4]{% many-to-many
1308   \@tempcnta=#1\relax
1309   \@tempcntb=#4\relax
1310   \def\bbl@tempa{%
1311     \ifnum\@tempcnta>#2\else
1312       \@expandtwoargs\BabelLower{\the\@tempcnta}{\the\@tempcntb}%
1313       \advance\@tempcnta#3\relax
1314       \advance\@tempcntb#3\relax
1315       \expandafter\bbl@tempa
1316     \fi}%
1317   \bbl@tempa}
1318 \newcommand\BabelLowerMO[4]{% many-to-one
1319   \@tempcnta=#1\relax
1320   \def\bbl@tempa{%
1321     \ifnum\@tempcnta>#2\else
1322       \@expandtwoargs\BabelLower{\the\@tempcnta}{#4}%
1323       \advance\@tempcnta#3
1324       \expandafter\bbl@tempa
1325     \fi}%
1326   \bbl@tempa}

```

## 7.17 Hooks

Note they are loaded in babel.def. switch.def only provides a “hook” for hooks (with a default value which is a no-op, below). Admittedly, the current implementation is a somewhat simplistic and does vety little to catch errors, but it is intended for developpers, after all. \bbl@usehooks is the commands used by babel to execute hooks defined for an event.

```

1327 \def\AddBabelHook#1#2{%
1328   \@ifundefined{bbl@hk@#1}{\EnableBabelHook{#1}}{}%
1329   \def\bbl@tempa##1,##2=##2,##3\@empty{\def\bbl@tempb{##2}}%
1330   \expandafter\bbl@tempa\bbl@evargs,##2=,\@empty
1331   \@ifundefined{bbl@ev@#1@#2}%
1332     {\bbl@csarg\bbl@add{ev@#2}{\bbl@elt{#1}}}%
1333     \bbl@csarg\newcommand}%
1334     {\bbl@csarg\renewcommand}%
1335     {ev@#1@#2}[\bbl@tempb]}
1336 \def\EnableBabelHook#1{\bbl@csarg\let{hk@#1}\@firstofone}
1337 \def\DisableBabelHook#1{\bbl@csarg\let{hk@#1}\@gobble}
1338 \def\bbl@usehooks#1#2{%

```

```

1339 \def\bbl@elt##1{%
1340   \@nameuse{bbl@hk@##1}{\@nameuse{bbl@ev@##1@#1}#2}}%
1341 \@nameuse{bbl@ev@#1}}

```

To ensure forward compatibility, arguments in hooks are set implicitly. So, if a further argument is added in the future, there is no need to change the existing code. Note events intended for hyphen.cfg are also loaded (just in case you need them for some reason).

```

1342 \def\bbl@evargs{,% don't delete the comma
1343   everylanguage=1,loadkernel=1,loadpatterns=1,loadexceptions=1,%
1344   adddialect=2,patterns=2,defaultcommands=0,encodedcommands=2,write=0,%
1345   beforeextras=0,afterextras=0,stopcommands=0,stringprocess=0,%
1346   hyphenation=2}
1347 \ifcase\bbl@engine\or
1348   \input luababel.def
1349 \or
1350   \input xebabel.def
1351 \fi

```

## 7.18 The redefinition of the style commands

The rest of the code in this file can only be processed by L<sup>A</sup>T<sub>E</sub>X, so we check the current format. If it is plain T<sub>E</sub>X, processing should stop here. But, because of the need to limit the scope of the definition of `\format`, a macro that is used locally in the following `\if` statement, this comparison is done inside a group. To prevent T<sub>E</sub>X from complaining about an unclosed group, the processing of the command `\endinput` is deferred until after the group is closed. This is accomplished by the command `\aftergroup`.

```

1352 {\def\format{lplain}
1353 \ifx\fmtname\format
1354 \else
1355   \def\format{LaTeX2e}
1356   \ifx\fmtname\format
1357   \else
1358     \aftergroup\endinput
1359   \fi
1360 \fi}

```

## 7.19 Cross referencing macros

The L<sup>A</sup>T<sub>E</sub>X book states:

The key argument is any sequence of letters, digits, and punctuation symbols; upper- and lowercase letters are regarded as different.

When the above quote should still be true when a document is typeset in a language that has active characters, special care has to be taken of the category codes of these characters when they appear in an argument of the cross referencing macros.

When a cross referencing command processes its argument, all tokens in this argument should be character tokens with category ‘letter’ or ‘other’.

The only way to accomplish this in most cases is to use the trick described in the T<sub>E</sub>Xbook [1] (Appendix D, page 382). The primitive `\meaning` applied to a token expands to the current meaning of this token. For example, `\meaning\A` with `\A`

defined as ‘`\def\A#1{\B}`’ expands to the characters ‘`macro:#1->\B`’ with all category codes set to ‘other’ or ‘space’.

`\newlabel` The macro `\label` writes a line with a `\newlabel` command into the `.aux` file to define labels.

```
1361 %\bbl@redefine\newlabel#1#2{%
1362 %  \@safe@activestrue\org@newlabel{#1}{#2}\@safe@activfalse}
```

`\@newl@bel` We need to change the definition of the L<sup>A</sup>T<sub>E</sub>X-internal macro `\@newl@bel`. This is needed because we need to make sure that shorthand characters expand to their non-active version.  
First we open a new group to keep the changed setting of `\protect` local and then we set the `@safe@actives` switch to true to make sure that any shorthand that appears in any of the arguments immediately expands to its non-active self.

```
1363 \ifx\bbl@opt@safe\@empty\else
1364   \def\@newl@bel#1#2#3{%
1365     {\@safe@activestrue
1366       \ifundefined{#1@#2}%
1367         \relax
1368         {\gdef\@multiplelabels{%
1369           \@latex@warning@no@line{There were multiply-defined labels}}}%
1370         \@latex@warning@no@line{Label ‘#2’ multiply defined}}%
1371       \global\@namedef{#1@#2}{#3}}}
```

`\@testdef` An internal L<sup>A</sup>T<sub>E</sub>X macro used to test if the labels that have been written on the `.aux` file have changed. It is called by the `\enddocument` macro. This macro needs to be completely rewritten, using `\meaning`. The reason for this is that in some cases the expansion of `#1@#2` contains the same characters as the `#3`; but the character codes differ. Therefore L<sup>A</sup>T<sub>E</sub>X keeps reporting that the labels may have changed.

```
1372 \CheckCommand*\@testdef[3]{%
1373   \def\reserved@a{#3}%
1374   \expandafter\ifx\csname#1@#2\endcsname\reserved@a
1375   \else
1376     \@tempswatrue
1377   \fi}
```

Now that we made sure that `\@testdef` still has the same definition we can rewrite it. First we make the shorthands ‘safe’.

```
1378 \def\@testdef#1#2#3{%
1379   \@safe@activestrue
```

Then we use `\bbl@tempa` as an ‘alias’ for the macro that contains the label which is being checked.

```
1380   \expandafter\let\expandafter\bbl@tempa\csname #1@#2\endcsname
```

Then we define `\bbl@tempb` just as `\@newl@bel` does it.

```
1381   \def\bbl@tempb{#3}%
1382   \@safe@activfalse
```

When the label is defined we replace the definition of `\bbl@tempa` by its meaning.

```
1383   \ifx\bbl@tempa\relax
1384   \else
1385     \edef\bbl@tempa{\expandafter\strip@prefix\meaning\bbl@tempa}%
1386   \fi
```



We do the same for `\bbl@tempb`.

```
1387 \edef\bbl@tempb{\expandafter\strip@prefix\meaning\bbl@tempb}%
```

If the label didn't change, `\bbl@tempa` and `\bbl@tempb` should be identical macros.

```
1388 \ifx\bbl@tempa\bbl@tempb
```

```
1389 \else
```

```
1390 \@tempswatru
```

```
1391 \fi}
```

```
1392 \fi
```

`\ref` The same holds for the macro `\ref` that references a label and `\pageref` to reference a page. So we redefine `\ref` and `\pageref`. While we change these macros, we make them robust as well (if they weren't already) to prevent problems if they should become expanded at the wrong moment.

```
1393 \@expandtwoargs\in@{R}\bbl@opt@safe
```

```
1394 \ifin@
```

```
1395 \bbl@redefineroobust\ref#1{%
```

```
1396 \@safe@activestrue\org@ref{#1}\@safe@activ>false}
```

```
1397 \bbl@redefineroobust\pageref#1{%
```

```
1398 \@safe@activestrue\org@pageref{#1}\@safe@activ>false}
```

```
1399 \else
```

```
1400 \let\org@ref\ref
```

```
1401 \let\org@pageref\pageref
```

```
1402 \fi
```

`\@citex` The macro used to cite from a bibliography, `\cite`, uses an internal macro, `\@citex`. It is this internal macro that picks up the argument(s), so we redefine this internal macro and leave `\cite` alone. The first argument is used for typesetting, so the shorthands need only be deactivated in the second argument.

```
1403 \@expandtwoargs\in@{B}\bbl@opt@safe
```

```
1404 \ifin@
```

```
1405 \bbl@redefine\@citex[#1]#2{%
```

```
1406 \@safe@activestrue\edef\@tempa{#2}\@safe@activ>false}
```

```
1407 \org@citex[#1]{\@tempa}}
```

Unfortunately, the packages `natbib` and `cite` need a different definition of `\@citex`... To begin with, `natbib` has a definition for `\@citex` with *three* arguments... We only know that a package is loaded when `\begin{document}` is executed, so we need to postpone the different redefinition.

```
1408 \AtBeginDocument{%
```

```
1409 \ifpackageloaded{natbib}{%
```

Notice that we use `\def` here instead of `\bbl@redefine` because `\org@citex` is already defined and we don't want to overwrite that definition (it would result in parameter stack overflow because of a circular definition).

(Recent versions of `natbib` change dynamically `\@citex`, so PR4087 doesn't seem fixable in a simple way. Just load `natbib` before.)

```
1410 \def\@citex[#1][#2]#3{%
```

```
1411 \@safe@activestrue\edef\@tempa{#3}\@safe@activ>false}
```

```
1412 \org@citex[#1][#2]{\@tempa}}%
```

```
1413 }{}}
```

The package `cite` has a definition of `\@citex` where the shorthands need to be turned off in both arguments.

```
1414 \AtBeginDocument{%
```

```
1415 \ifpackageloaded{cite}{%
```

```

1416      \def\@citex[#1]#2{%
1417          \@safe@activestrue\org@@citex[#1]{#2}\@safe@activesfalse}%
1418      }{}}

\nocite The macro \nocite which is used to instruct BiBTeX to extract uncited references
        from the database.

1419      \bbl@redefine\nocite#1{%
1420          \@safe@activestrue\org@nocite{#1}\@safe@activesfalse}

\bibcite The macro that is used in the .aux file to define citation labels. When packages
        such as natbib or cite are not loaded its second argument is used to typeset the
        citation label. In that case, this second argument can contain active characters but
        is used in an environment where \@safe@activestrue is in effect. This switch
        needs to be reset inside the \hbox which contains the citation label. In order to
        determine during .aux file processing which definition of \bibcite is needed we
        define \bibcite in such a way that it redefines itself with the proper definition.
1421      \bbl@redefine\bibcite{%
        We call \bbl@cite@choice to select the proper definition for \bibcite. This new
        definition is then activated.
1422          \bbl@cite@choice
1423          \bibcite}

\bbl@bibcite The macro \bbl@bibcite holds the definition of \bibcite needed when neither
        natbib nor cite is loaded.
1424      \def\bbl@bibcite#1#2{%
1425          \org@bibcite{#1}{\@safe@activesfalse#2}}

\bbl@cite@choice The macro \bbl@cite@choice determines which definition of \bibcite is needed.
1426      \def\bbl@cite@choice{%
        First we give \bibcite its default definition.
1427          \global\let\bibcite\bbl@bibcite
        Then, when natbib is loaded we restore the original definition of \bibcite.
1428          \@ifpackageloaded{natbib}{\global\let\bibcite\org@bibcite}{}%
        For cite we do the same.
1429          \@ifpackageloaded{cite}{\global\let\bibcite\org@bibcite}{}%
        Make sure this only happens once.
1430          \global\let\bbl@cite@choice\relax
        When a document is run for the first time, no .aux file is available, and \bibcite
        will not yet be properly defined. In this case, this has to happen before the
        document starts.
1431      \AtBeginDocument{\bbl@cite@choice}

\@bibitem One of the two internal LATEX macros called by \bibitem that write the citation
        label on the .aux file.
1432      \bbl@redefine\@bibitem#1{%
1433          \@safe@activestrue\org@@bibitem{#1}\@safe@activesfalse}
1434      \else
1435          \let\org@nocite\nocite
1436          \let\org@@citex\@citex
1437          \let\org@bibcite\bibcite
1438          \let\org@@bibitem\@bibitem
1439      \fi

```

## 7.20 Marks

`\markright` Because the output routine is asynchronous, we must pass the current language attribute to the head lines, together with the text that is put into them. To achieve this we need to adapt the definition of `\markright` and `\markboth` somewhat.

```
1440 \bbl@redefine\markright#1{%
```

First of all we temporarily store the language switching command, using an expanded definition in order to get the current value of `\language`.

```
1441 \edef\bbl@tempb{\noexpand\protect
```

```
1442 \noexpand\foreignlanguage{\language}}}%
```

Then, we check whether the argument is empty; if it is, we just make sure the scratch token register is empty.

```
1443 \def\bbl@arg{#1}%
```

```
1444 \ifx\bbl@arg@empty
```

```
1445 \toks@{}%
```

```
1446 \else
```

Next, we store the argument to `\markright` in the scratch token register, together with the expansion of `\bbl@tempb` (containing the language switching command) as defined before. This way these commands will not be expanded by using `\edef` later on, and we make sure that the text is typeset using the correct language settings. While doing so, we make sure that active characters that may end up in the mark are not disabled by the output routine kicking in while `\@safe@activestrue` is in effect.

```
1447 \expandafter\toks@\expandafter{%
```

```
1448 \bbl@tempb{\protect\bbl@restore@actives#1}}%
```

```
1449 \fi
```

Then we define a temporary control sequence using `\edef`.

```
1450 \edef\bbl@tempa{%
```

When `\bbl@tempa` is executed, only `\language` will be expanded, because of the way the token register was filled.

```
1451 \noexpand\org@markright{\the\toks@}}%
```

```
1452 \bbl@tempa
```

```
1453 }
```

`\markboth` The definition of `\markboth` is equivalent to that of `\markright`, except that we  
`\@mkboth` need two token registers. The documentclasses `report` and `book` define and set the headings for the page. While doing so they also store a copy of `\markboth` in `\@mkboth`. Therefore we need to check whether `\@mkboth` has already been set. If so we need to do that again with the new definition of `\makrboth`.

```
1454 \ifx\@mkboth\markboth
```

```
1455 \def\bbl@tempc{\let\@mkboth\markboth}
```

```
1456 \else
```

```
1457 \def\bbl@tempc{}
```

```
1458 \fi
```

Now we can start the new definition of `\markboth`

```
1459 \bbl@redefine\markboth#1#2{%
```

```
1460 \edef\bbl@tempb{\noexpand\protect
```

```
1461 \noexpand\foreignlanguage{\language}}}%
```

```
1462 \def\bbl@arg{#1}%
```

```
1463 \ifx\bbl@arg@empty
```

```
1464 \toks@{}%
```

```

1465 \else
1466 \expandafter\toks@\expandafter{%
1467     \bbl@tempb{\protect\bbl@restore@actives#1}}%
1468 \fi
1469 \def\bbl@arg{#2}%
1470 \ifx\bbl@arg@empty
1471     \toks8{}%
1472 \else
1473     \expandafter\toks8\expandafter{%
1474         \bbl@tempb{\protect\bbl@restore@actives#2}}%
1475 \fi
1476 \edef\bbl@tempa{%
1477     \noexpand\org@markboth{\the\toks@}{\the\toks8}}%
1478 \bbl@tempa
1479 }

```

and copy it to \@mkboth if necessary.

```

1480 \bbl@tempc

```

## 7.21 Preventing clashes with other packages

### 7.21.1 ifthen

`\ifthenelse` Sometimes a document writer wants to create a special effect depending on the page a certain fragment of text appears on. This can be achieved by the following piece of code:

```

% \ifthenelse{\isodd{\pageref{some:label}}}{
%     {code for odd pages}
%     {code for even pages}
%

```

In order for this to work the argument of `\isodd` needs to be fully expandable. With the above redefinition of `\pageref` it is not in the case of this example. To overcome that, we add some code to the definition of `\ifthenelse` to make things work.

The first thing we need to do is check if the package `ifthen` is loaded. This should be done at `\begin{document}` time.

```

1481 \@expandtwoargs\in@{R}\bbl@opt@safe
1482 \ifin@
1483     \AtBeginDocument{%
1484         \@ifpackageloaded{ifthen}{%

```

Then we can redefine `\ifthenelse`:

```

1485     \bbl@redefine@long\ifthenelse#1#2#3{%

```

We want to revert the definition of `\pageref` and `\ref` to their original definition for the duration of `\ifthenelse`, so we first need to store their current meanings.

```

1486     \let\bbl@tempa\pageref
1487     \let\pageref\org@pageref
1488     \let\bbl@tempb\ref
1489     \let\ref\org@ref

```

Then we can set the `\@safe@actives` switch and call the original `\ifthenelse`. In order to be able to use shorthands in the second and third arguments of `\ifthenelse` the resetting of the switch *and* the definition of `\pageref` happens inside those arguments. When the package wasn't loaded we do nothing.

```

1490      \@safe@activestru
1491      \org@ifthenelse{#1}{%
1492      \let\pageref\bbl@tempa
1493      \let\ref\bbl@tempb
1494      \@safe@activestru
1495      #2}{%
1496      \let\pageref\bbl@tempa
1497      \let\ref\bbl@tempb
1498      \@safe@activestru
1499      #3}%
1500    }%
1501  }{}%
1502 }

```

### 7.21.2 varioref

`\@vpageref` When the package `varioref` is in use we need to modify its internal command `\vrefpagemum` `\@vpageref` in order to prevent problems when an active character ends up in the `\Ref` argument of `\vref`.

```

1503 \AtBeginDocument{%
1504   \@ifpackageloaded{varioref}{%
1505     \bbl@redefine\@vpageref#1[#2]#3{%
1506       \@safe@activestru
1507       \org\@vpageref{#1}[#2]#3}%
1508     \@safe@activestru

```

The same needs to happen for `\vrefpagemum`.

```

1509     \bbl@redefine\vrefpagemum#1#2{%
1510       \@safe@activestru
1511       \org\vrefpagemum{#1}#2}%
1512     \@safe@activestru

```

The package `varioref` defines `\Ref` to be a robust command which uppercases the first character of the reference text. In order to be able to do that it needs to access the expandable form of `\ref`. So we employ a little trick here. We redefine the (internal) command `\Ref_` to call `\org@ref` instead of `\ref`. The disadvantage of this solution is that whenever the definition of `\Ref` changes, this definition needs to be updated as well.

```

1513     \expandafter\def\csname Ref \endcsname#1{%
1514       \protected@edef\@tempa{\org@ref{#1}}\expandafter\MakeUppercase\@tempa}
1515     }{}%
1516   }
1517 \fi

```

### 7.21.3 hhline

`\hhline` Delaying the activation of the shorthand characters has introduced a problem with the `hhline` package. The reason is that it uses the `‘:` character which is made active by the french support in `babel`. Therefore we need to *reload* the package when the `‘:` is an active character.

So at `\begin{document}` we check whether `hhline` is loaded.

```

1518 \AtEndOfPackage{%
1519   \AtBeginDocument{%
1520     \@ifpackageloaded{hhline}%

```

Then we check whether the expansion of `\normal@char:` is not equal to `\relax`.

```
1521      {\expandafter\ifx\csname normal@char\string:\endcsname\relax
1522      \else
```

In that case we simply reload the package. Note that this happens *after* the category code of the `@`-sign has been changed to other, so we need to temporarily change it to letter again.

```
1523      \makeatletter
1524      \def\@currname{hhline}\input{hhline.sty}\makeatother
1525      \fi}%
1526      {}}}
```

#### 7.21.4 hyperref

`\pdfstringdefDisableCommands` A number of interworking problems between `babel` and `hyperref` are tackled by `hyperref` itself. The following code was introduced to prevent some annoying warnings but it broke bookmarks. This was quickly fixed in `hyperref`, which essentially made it no-op. However, it will not be removed for the moment because `hyperref` is expecting it.

```
1527 \AtBeginDocument{%
1528   \@ifundefined{pdfstringdefDisableCommands}%
1529   {}%
1530   {\pdfstringdefDisableCommands{%
1531     \languageshorthands{system}}}%
1532   }%
1533 }
```

#### 7.21.5 fancyhdr

`\FOREIGNLANGUAGE` The package `fancyhdr` treats the running head and foot lines somewhat differently as the standard classes. A symptom of this is that the command `\foreignlanguage` which `babel` adds to the marks can end up inside the argument of `\MakeUppercase`. To prevent unexpected results we need to define `\FOREIGNLANGUAGE` here.

```
1534 \DeclareRobustCommand{\FOREIGNLANGUAGE}[1]{%
1535   \lowercase{\foreignlanguage{#1}}}
```

`\substitutefontfamily` The command `\substitutefontfamily` creates an `.fd` file on the fly. The first argument is an encoding mnemonic, the second and third arguments are font family names.

```
1536 \def\substitutefontfamily#1#2#3{%
1537   \lowercase{\immediate\openout15=#1#2.fd\relax}%
1538   \immediate\write15{%
1539     \string\ProvidesFile{#1#2.fd}%
1540     [\the\year/\two@digits{\the\month}/\two@digits{\the\day}
1541     \space generated font description file]^J
1542     \string\DeclareFontFamily{#1}{#2}{ }^^J
1543     \string\DeclareFontShape{#1}{#2}{m}{n}{<->ssub * #3/m/n}{ }^^J
1544     \string\DeclareFontShape{#1}{#2}{m}{it}{<->ssub * #3/m/it}{ }^^J
1545     \string\DeclareFontShape{#1}{#2}{m}{sl}{<->ssub * #3/m/sl}{ }^^J
1546     \string\DeclareFontShape{#1}{#2}{m}{sc}{<->ssub * #3/m/sc}{ }^^J
1547     \string\DeclareFontShape{#1}{#2}{b}{n}{<->ssub * #3/bx/n}{ }^^J
1548     \string\DeclareFontShape{#1}{#2}{b}{it}{<->ssub * #3/bx/it}{ }^^J
1549     \string\DeclareFontShape{#1}{#2}{b}{sl}{<->ssub * #3/bx/sl}{ }^^J
1550     \string\DeclareFontShape{#1}{#2}{b}{sc}{<->ssub * #3/bx/sc}{ }^^J
```

```

1551     }%
1552   \closeout15
1553 }

```

This command should only be used in the preamble of a document.

```

1554 \@onlypreamble\substitutefontfamily

```

## 7.22 Encoding issues (part 2)

`\TeX` Because documents may use font encodings other than one of the latin encodings,  
`\LaTeX` we make sure that the logos of `TEX` and `LATEX` always come out in the right encoding.

```

1555 \bbl@redefine\TeX{\textlatin{\org@TeX}}
1556 \bbl@redefine\LaTeX{\textlatin{\org@LaTeX}}

```

`\nfss@catcodes` `LATEX`'s font selection scheme sometimes wants to read font definition files in the middle of processing the document. In order to guard against any characters having the wrong `\catcodes` it always calls `\nfss@catcodes` before loading a file. Unfortunately, the characters " and ' are not dealt with. Therefore we have to add them until `LATEX` does that itself.

```

1557 \bbl@add\nfss@catcodes{%
1558   \@makeother\'%
1559   \@makeother\"}

```

## 7.23 Local Language Configuration

`\loadlocalcfg` At some sites it may be necessary to add site-specific actions to a language definition file. This can be done by creating a file with the same name as the language definition file, but with the extension `.cfg`. For instance the file `norsk.cfg` will be loaded when the language definition file `norsk.ldf` is loaded. For plain-based formats we don't want to override the definition of `\loadlocalcfg` from `plain.def`.

```

1560 \ifx\loadlocalcfg\@undefined
1561   \@ifpackagewith{babel}{noconfigs}%
1562     {\let\loadlocalcfg@gobble}%
1563     {\def\loadlocalcfg#1{%
1564       \InputIfFileExists{#1.cfg}%
1565       {\typeout{*****^J%
1566                * Local config file #1.cfg used^^J%
1567                *}}%
1568       \@empty}}
1569 \fi

```

Just to be compatible with `LATEX 2.09` we add a few more lines of code:

```

1570 \ifx\@unexpandable@protect\@undefined
1571   \def\@unexpandable@protect{\noexpand\protect\noexpand}
1572   \long\def\protected@write#1#2#3{%
1573     \begingroup
1574       \let\thepage\relax
1575       #2%
1576       \let\protect\@unexpandable@protect
1577       \edef\reserved@a{\write#1{#3}}%
1578       \reserved@a
1579     \endgroup
1580     \if@nobreak\ifvmode\nobreak\fi\fi}

```

```
1581 \fi
```

Finally, the default is to use English as the main language.

```
1582 \ifx\l@english\@undefined
```

```
1583   \chardef\l@english\z@
```

```
1584 \fi
```

```
1585 \main@language{english}
```

```
1586 \</core>
```

Now that we're sure that the code is seen by L<sup>A</sup>T<sub>E</sub>X only, we have to find out what the main (primary) document style is because we want to redefine some macros. This is only necessary for releases of L<sup>A</sup>T<sub>E</sub>X dated before December 1991. Therefore this part of the code can optionally be included in `babel.def` by specifying the `docstrip` option names.

```
1587 (*names)
```

The standard styles can be distinguished by checking whether some macros are defined. In table 1 an overview is given of the macros that can be used for this purpose.

article	:	both the <code>\chapter</code> and <code>\opening</code> macros are undefined
report and book	:	the <code>\chapter</code> macro is defined and the <code>\opening</code> is undefined
letter	:	the <code>\chapter</code> macro is undefined and the <code>\opening</code> is defined

Table 1: How to determine the main document style

The macros that have to be redefined for the `report` and `book` document styles happen to be the same, so there is no need to distinguish between those two styles.

`\doc@style` First a parameter `\doc@style` is defined to identify the current document style. This parameter might have been defined by a document style that already uses macros instead of hard-wired texts, such as `artikel1.sty` [6], so the existence of `\doc@style` is checked. If this macro is undefined, i. e., if the document style is unknown and could therefore contain hard-wired texts, `\doc@style` is defined to the default value '0'.

```
1588 \ifx\@undefined\doc@style
```

```
1589   \def\doc@style{0}%
```

This parameter is defined in the following `if` construction (see table 1):

```
1590   \ifx\@undefined\opening
```

```
1591     \ifx\@undefined\chapter
```

```
1592       \def\doc@style{1}%
```

```
1593     \else
```

```
1594       \def\doc@style{2}%
```

```
1595     \fi
```

```
1596   \else
```

```
1597     \def\doc@style{3}%
```

```
1598   \fi%
```

```
1599 \fi%
```



### 7.23.1 Redefinition of macros

Now here comes the real work: we start to redefine things and replace hard-wired texts by macros. These redefinitions should be carried out conditionally, in case it has already been done.

For the figure and table environments we have in all styles:

```
1600 \ifundefined{figurename}{\def\fnun@figure{figurename} \thefigure}}{}
1601 \ifundefined{tablename}{\def\fnun@table{tablename} \thetable}}{}
```

The rest of the macros have to be treated differently for each style. When `\doc@style` still has its default value nothing needs to be done.

```
1602 \ifcase \doc@style\relax
1603 \or
```

This means that `babel.def` is read after the article style, where no `\chapter` and `\opening` commands are defined<sup>21</sup>.

First we have the `\tableofcontents`, `\listoffigures` and `\listoftables`:

```
1604 \ifundefined{contentsname}%
1605   {\def\tableofcontents{\section*{\contentsname\mkboth
1606     {\uppercase{\contentsname}}{\uppercase{\contentsname}}}%
1607     \@starttoc{toc}}}{}}
1608 \ifundefined{listfigurename}%
1609   {\def\listoffigures{\section*{\listfigurename\mkboth
1610     {\uppercase{\listfigurename}}{\uppercase{\listfigurename}}}%
1611     \@starttoc{lof}}}{}}
1612 \ifundefined{listtablename}%
1613   {\def\listoftables{\section*{\listtablename\mkboth
1614     {\uppercase{\listtablename}}{\uppercase{\listtablename}}}%
1615     \@starttoc{lot}}}{}}
```

Then the `\thebibliography` and `\theindex` environments.

```
1616 \ifundefined{refname}%
1617   {\def\thebibliography#1{\section*{\refname
1618     \mkboth{\uppercase{\refname}}{\uppercase{\refname}}}%
1619     \list{[arabic{enumi}]}{\settowidth\labelwidth{#1}%
1620       \leftmargin\labelwidth
1621       \advance\leftmargin\labelsep
1622       \usecounter{enumi}}%
1623     \def\newblock{\hskip.11em plus.33em minus.07em}%
1624     \sloppy\clubpenalty4000\widowpenalty\clubpenalty
1625     \sfcode'\.=1000\relax}}{}}
1626 \ifundefined{indexname}%
1627   {\def\theindex{\@restonecoltrue\if@twocolumn\@restonecolfalse\fi
1628     \columnseprule \z@
1629     \columnsep 35pt\twocolumn[\section*{\indexname}%
1630       \mkboth{\uppercase{\indexname}}{\uppercase{\indexname}}}%
1631     \thispagestyle{plain}%
1632     \parskip\z@ plus.3pt\parindent\z@\let\item\@idxitem}}{}}
```

The abstract environment:

```
1633 \ifundefined{abstractname}%
1634   {\def\abstract{\if@twocolumn
1635     \section*{\abstractname}%
1636     \else \small
```

---

<sup>21</sup>A fact that was pointed out to me by Nico Poppelier and was already used in Piet van Oostrum's document style option `nl`.

```

1637 \begin{center}%
1638 {\bf \abstractname\vspace{-.5em}\vspace{\z@}}%
1639 \end{center}%
1640 \quotation
1641 \fi}}{}

```

And last but not least, the macro \part:

```

1642 \@ifundefined{partname}%
1643 {\def\@part[#1]#2{\ifnum \c@secnumdepth >\m@ne
1644 \refstepcounter{part}%
1645 \addcontentsline{toc}{part}{\thepart
1646 \hspace{1em}#1}\else
1647 \addcontentsline{toc}{part}{#1}\fi
1648 {\parindent\z@ \raggedright
1649 \ifnum \c@secnumdepth >\m@ne
1650 \Large \bf \partname{} \thepart
1651 \par \nobreak
1652 \fi
1653 \huge \bf
1654 #2\markboth{}{}\par}%
1655 \nobreak
1656 \vskip 3ex\@afterheading}%
1657 }{}

```

This is all that needs to be done for the article style.

```

1658 \or

```

The next case is formed by the two styles book and report. Basically we have to do the same as for the article style, except now we must also change the \chapter command.

The tables of contents, figures and tables:

```

1659 \@ifundefined{contentsname}%
1660 {\def\tableofcontents{\@restonecolfalse
1661 \if@twocolumn\@restonecoltrue\onecolumn
1662 \fi\chapter*{\contentsname\@mkboth
1663 {\uppercase{\contentsname}}{\uppercase{\contentsname}}}%
1664 \@starttoc{toc}%
1665 \csname if@restonecol\endcsname\twocolumn
1666 \csname fi\endcsname}}{}
1667 \@ifundefined{listfigurename}%
1668 {\def\listoffigures{\@restonecolfalse
1669 \if@twocolumn\@restonecoltrue\onecolumn
1670 \fi\chapter*{\listfigurename\@mkboth
1671 {\uppercase{\listfigurename}}{\uppercase{\listfigurename}}}%
1672 \@starttoc{lof}%
1673 \csname if@restonecol\endcsname\twocolumn
1674 \csname fi\endcsname}}{}
1675 \@ifundefined{listtablename}%
1676 {\def\listoftables{\@restonecolfalse
1677 \if@twocolumn\@restonecoltrue\onecolumn
1678 \fi\chapter*{\listtablename\@mkboth
1679 {\uppercase{\listtablename}}{\uppercase{\listtablename}}}%
1680 \@starttoc{lot}%
1681 \csname if@restonecol\endcsname\twocolumn
1682 \csname fi\endcsname}}{}

```

Again, the bibliography and index environments; notice that in this case we use `\bibname` instead of `\refname` as in the definitions for the article style. The reason for this is that in the article document style the term ‘References’ is used in the definition of `\thebibliography`. In the report and book document styles the term ‘Bibliography’ is used.

```

1683 \ifundefined{bibname}%
1684   {\def\thebibliography#1{\chapter*{\bibname
1685     \mkboth{\uppercase{\bibname}}{\uppercase{\bibname}}}%
1686     \list{[\arabic{enumi}]}{\settowidth\labelwidth{[#1]}%
1687     \leftmargin\labelwidth \advance\leftmargin\labelsep
1688     \usecounter{enumi}}%
1689     \def\newblock{\hskip.11em plus.33em minus.07em}%
1690     \sloppy\clubpenalty4000\widowpenalty\clubpenalty
1691     \sfcode'\.=1000\relax}}{}
1692 \ifundefined{indexname}%
1693   {\def\theindex{\@restonecoltrue\if@twocolumn\@restonecolfalse\fi
1694     \columnseprule \z@
1695     \columnsep 35pt\twocolumn[\@makeschapterhead{\indexname}]%
1696     \mkboth{\uppercase{\indexname}}{\uppercase{\indexname}}%
1697     \thispagestyle{plain}%
1698     \parskip\z@ plus.3pt\parindent\z@ \let\item\idxitem}}{}

```

Here is the abstract environment:

```

1699 \ifundefined{abstractname}%
1700   {\def\abstract{\titlepage
1701     \null\vfil
1702     \begin{center}%
1703     {\bf \abstractname}%
1704     \end{center}}}{ }

```

And last but not least the `\chapter`, `\appendix` and `\part` macros.

```

1705 \ifundefined{chaptername}{\def\@chapapp{\chaptername}}{}
1706 %
1707 \ifundefined{appendixname}%
1708   {\def\appendix{\par
1709     \setcounter{chapter}{0}%
1710     \setcounter{section}{0}%
1711     \def\@chapapp{\appendixname}%
1712     \def\thechapter{\Alph{chapter}}}{ }}
1713 %
1714 \ifundefined{partname}%
1715   {\def\@part[#1]#2{\ifnum \c@secnumdepth >-2\relax
1716     \refstepcounter{part}%
1717     \addcontentsline{toc}{part}{\thepart
1718     \hspace{1em}#1}\else
1719     \addcontentsline{toc}{part}{#1}\fi
1720     \markboth{}{ }%
1721     {\centering
1722     \ifnum \c@secnumdepth >-2\relax
1723       \huge\bf \partname{ } \thepart
1724       \par
1725       \vskip 20pt \fi
1726       \Huge \bf
1727       #1\par}\@endpart}}{}%
1728 \or

```

Now we address the case where `babel.def` is read after the `letter` style. The `letter` document style defines the macro `\opening` and some other macros that are specific to `letter`. This means that we have to redefine other macros, compared to the previous two cases.

First two macros for the material at the end of a letter, the `\cc` and `\encl` macros.

```

1729 \@ifundefined{ccname}%
1730   {\def\cc#1{\par\noindent
1731     \parbox[t]{\textwidth}%
1732     {\@hangfrom{\rm \ccname : }\ignorespaces #1\strut}\par}}{}
1733 \@ifundefined{enclname}%
1734   {\def\encl#1{\par\noindent
1735     \parbox[t]{\textwidth}%
1736     {\@hangfrom{\rm \enclname : }\ignorespaces #1\strut}\par}}{}

```

The last thing we have to do here is to redefine the headings pagestyle:

```

1737 \@ifundefined{headtoname}%
1738   {\def\ps@headings{%
1739     \def\@oddhead{\sl \headtoname} \ignorespaces\toname \hfil
1740     \@date \hfil \pagename{} \thepage}%
1741     \def\@oddfoot{}}{}

```

This was the last of the four standard document styles, so if `\doc@style` has another value we do nothing and just close the `if` construction.

```

1742 \fi
1743 \</names>

```

Here ends the code that can be optionally included when a version of  $\text{\LaTeX}$  is in use that is dated *before* December 1991.

We also need to redefine a number of commands to ensure that the right font encoding is used, but this can't be done before `babel.def` is loaded.

## 7.24 Multiple languages

Plain  $\text{\TeX}$  version 3.0 provides the primitive `\language` that is used to store the current language. When used with a pre-3.0 version this function has to be implemented by allocating a counter.

```

1744 <*kernel>
1745 <<Make sure ProvidesFile is defined>>
1746 \ProvidesFile{switch.def}[<<date>>] <<version>> Babel switching mechanism]
1747 <<Load macros for plain if not LaTeX>>
1748 <<Define core switching macros>>

```

`\adddialect` The macro `\adddialect` can be used to add the name of a dialect or variant language, for which an already defined hyphenation table can be used.

```

1749 \def\bbl@version{<<version>>}
1750 \def\bbl@date{<<date>>}
1751 \def\adddialect#1#2{%
1752   \global\chardef#1#2\relax
1753   \bbl@usehooks{adddialect}{#1}{#2}}%
1754   \wlog{\string#1 = a dialect from \string\language#2}

```

`\bbl@iflanguage` executes code only if the language exists. Otherwise raises an error.

The argument of `\bbl@fixname` has to be a macro name, as it may get “fixed” if casing (lc/uc) is wrong. It's intended to fix a long-standing bug when

`\foreignlanguage` and the like appear in a `\MakeXXXcase`. However, a lowercase form is not imposed to improve backward compatibility (perhaps you defined a language named MYLANG, but unfortunately mixed case names cannot be trapped).

```

1755 \def\bbl@fixname#1{%
1756   \begingroup
1757   \def\bbl@tempe{l}%
1758   \edef\bbl@tempd{\noexpand\@ifundefined{\noexpand\bbl@tempe#1}}%
1759   \bbl@tempd
1760   {\lowercase\expandafter{\bbl@tempd}%
1761    {\uppercase\expandafter{\bbl@tempd}%
1762     \@empty
1763     {\edef\bbl@tempd{\def\noexpand#1{#1}}%
1764      \uppercase\expandafter{\bbl@tempd}}}%
1765   {\edef\bbl@tempd{\def\noexpand#1{#1}}%
1766    \lowercase\expandafter{\bbl@tempd}}}%
1767   \@empty
1768   \edef\bbl@tempd{\endgroup\def\noexpand#1{#1}}%
1769   \bbl@tempd}
1770 \def\bbl@iflanguage#1{%
1771   \@ifundefined{l@#1}{\@nolanerr{#1}\@gobble}\@firstofone}

```

`\iflanguage` Users might want to test (in a private package for instance) which language is currently active. For this we provide a test macro, `\iflanguage`, that has three arguments. It checks whether the first argument is a known language. If so, it compares the first argument with the value of `\language`. Then, depending on the result of the comparison, it executes either the second or the third argument.

```

1772 \def\iflanguage#1{%
1773   \bbl@iflanguage{#1}{%
1774     \ifnum\csname l@#1\endcsname=\language
1775       \expandafter\@firstoftwo
1776     \else
1777       \expandafter\@secondoftwo
1778     \fi}}

```

`\selectlanguage` The macro `\selectlanguage` checks whether the language is already defined before it performs its actual task, which is to update `\language` and activate language-specific definitions.

To allow the call of `\selectlanguage` either with a control sequence name or with a simple string as argument, we have to use a trick to delete the optional escape character.

To convert a control sequence to a string, we use the `\string` primitive. Next we have to look at the first character of this string and compare it with the escape character. Because this escape character can be changed by setting the internal integer `\escapechar` to a character number, we have to compare this number with the character of the string. To do this we have to use T<sub>E</sub>X's backquote notation to specify the character as a number.

If the first character of the `\string`'ed argument is the current escape character, the comparison has stripped this character and the rest in the 'then' part consists of the rest of the control sequence name. Otherwise we know that either the argument is not a control sequence or `\escapechar` is set to a value outside of the character range 0–255.

If the user gives an empty argument, we provide a default argument for `\string`. This argument should expand to nothing.

```

1779 \let\bbl@select@type\z@
1780 \edef\selectlanguage{%
1781   \noexpand\protect
1782   \expandafter\noexpand\csname selectlanguage \endcsname}

```

Because the command `\selectlanguage` could be used in a moving argument it expands to `\protect\selectlanguage_`. Therefore, we have to make sure that a macro `\protect` exists. If it doesn't it is `\let` to `\relax`.

```

1783 \ifx\@undefined\protect\let\protect\relax\fi

```

As L<sup>A</sup>T<sub>E</sub>X 2.09 writes to files *expanded* whereas L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> takes care *not* to expand the arguments of `\write` statements we need to be a bit clever about the way we add information to .aux files. Therefore we introduce the macro `\xstring` which should expand to the right amount of `\string`'s.

```

1784 \ifx\documentclass\@undefined
1785   \def\xstring{\string\string\string}
1786 \else
1787   \let\xstring\string
1788 \fi

```

Since version 3.5 `babel` writes entries to the auxiliary files in order to typeset table of contents etc. in the correct language environment.

`\bbl@pop@language` But when the language change happens *inside* a group the end of the group doesn't write anything to the auxiliary files. Therefore we need T<sub>E</sub>X's `aftergroup` mechanism to help us. The command `\aftergroup` stores the token immediately following it to be executed when the current group is closed. So we define a temporary control sequence `\bbl@pop@language` to be executed at the end of the group. It calls `\bbl@set@language` with the name of the current language as its argument.

`\bbl@language@stack` The previous solution works for one level of nesting groups, but as soon as more levels are used it is no longer adequate. For that case we need to keep track of the nested languages using a stack mechanism. This stack is called `\bbl@language@stack` and initially empty.

```

1789 \def\bbl@language@stack{}

```

When using a stack we need a mechanism to push an element on the stack and to retrieve the information afterwards.

`\bbl@push@language` The stack is simply a list of languagenames, separated with a '+' sign; the push  
`\bbl@pop@language` function can be simple:

```

1790 \def\bbl@push@language{%
1791   \xdef\bbl@language@stack{\languagename+\bbl@language@stack}

```

Retrieving information from the stack is a little bit less simple, as we need to remove the element from the stack while storing it in the macro `\languagename`. For this we first define a helper function.

`\bbl@pop@lang` This macro stores its first element (which is delimited by the '+'-sign) in `\languagename` and stores the rest of the string (delimited by '-') in its third argument.

```

1792 \def\bbl@pop@lang#1+#2-#3{%
1793   \edef\languagename{#1}\xdef#3{#2}}

```

The reason for the somewhat weird arrangement of arguments to the helper function is the fact it is called in the following way. This means that before `\bbl@pop@lang` is executed  $\TeX$  first *expands* the stack, stored in `\bbl@language@stack`. The result of that is that the argument string of `\bbl@pop@lang` contains one or more language names, each followed by a ‘+’-sign (zero language names won’t occur as this macro will only be called after something has been pushed on the stack) followed by the ‘-’-sign and finally the reference to the stack.

```
1794 \def\bbl@pop@language{%
1795   \expandafter\bbl@pop@lang\bbl@language@stack-\bbl@language@stack
1796   \expandafter\bbl@set@language\expandafter{\language}%}
```

Once the name of the previous language is retrieved from the stack, it is fed to `\bbl@set@language` to do the actual work of switching everything that needs switching.

```
1797 \expandafter\def\csname selectlanguage \endcsname#1{%
1798   \ifnum\bbl@hymapsel=\@ccclv\let\bbl@hymapsel\tw@fi
1799   \bbl@push@language
1800   \aftergroup\bbl@pop@language
1801   \bbl@set@language{#1}}
```

`\bbl@set@language` The macro `\bbl@set@language` takes care of switching the language environment *and* of writing entries on the auxiliary files. For historical reasons, language names can be either `language` or `\language`. To catch either form a trick is used, but unfortunately as a side effect the catcodes of letters in `\language` are not well defined. The list of auxiliary files can be extended by redefining `\BabelContentsFiles`, but make sure they are loaded inside a group (as `aux`, `toc`, `lof`, and `lot` do) or the last language of the document will remain active afterwards.

We also write a command to change the current language in the auxiliary files.

```
1802 \def\BabelContentsFiles{toc,lof,lot}%
1803 \def\bbl@set@language#1{%
1804   \edef\language{#1}%
1805   \ifnum\escapechar=\expandafter'\string#1\@empty
1806     \else\string#1\@emptyfi}%
1807   \select@language{\language}%
1808   \expandafter\ifx\csname date\language\endcsname\relax\else
1809     \if@filesw
1810       \protected@write\@auxout{\string\select@language{\language}}%
1811       \bbl@for\bbl@tempa\BabelContentsFiles{%
1812         \addtocontents{\bbl@tempa}{\xstring\select@language{\language}}}%
1813       \bbl@usehooks{write}}}%
1814   \fi
1815   \fi}
1816 \def\select@language#1{%
1817   \ifnum\bbl@hymapsel=\@ccclv\chardef\bbl@hymapsel4\relax\fi
1818   \edef\language{#1}%
1819   \bbl@fixname\language
1820   \bbl@iflanguage\language{%
1821     \expandafter\ifx\csname date\language\endcsname\relax
1822       \bbl@error
1823       {You haven't loaded the language #1\space yet}%
1824       {You may proceed, but expect unexpected results}%
1825     \else
```

```

1826      \let\bbl@select@type\z@
1827      \expandafter\bbl@switch\expandafter{\language}%
1828      \fi}}
1829% A bit of optimization:
1830 \def\select@language@x#1{%
1831   \ifcase\bbl@select@type
1832     \bbl@ifsamestring\language{#1}{\select@language{#1}}%
1833   \else
1834     \select@language{#1}%
1835   \fi}

```

First, check if the user asks for a known language. If so, update the value of `\language` and call `\originalTeX` to bring  $\TeX$  in a certain pre-defined state.

The name of the language is stored in the control sequence `\language`. Then we have to redefine `\originalTeX` to compensate for the things that have been activated. To save memory space for the macro definition of `\originalTeX`, we construct the control sequence name for the `\noextras<lang>` command at definition time by expanding the `\csname` primitive.

Now activate the language-specific definitions. This is done by constructing the names of three macros by concatenating three words with the argument of `\selectlanguage`, and calling these macros.

The switching of the values of `\lefthyphenmin` and `\righthyphenmin` is somewhat different. First we save their current values, then we check if `\<lang>hyphenmins` is defined. If it is not, we set default values (2 and 3), otherwise the values in `\<lang>hyphenmins` will be used.

```

1836 \def\bbl@switch#1{%
1837   \originalTeX
1838   \expandafter\def\expandafter\originalTeX\expandafter{%
1839     \csname noextras#1\endcsname
1840     \let\originalTeX\@empty
1841     \babel@beginsave}%
1842   \languageshorthands{none}%
1843   \ifcase\bbl@select@type
1844     \csname captions#1\endcsname
1845     \csname date#1\endcsname
1846   \fi
1847   \bbl@usehooks{beforeextras}{}%
1848   \csname extras#1\endcsname\relax
1849   \bbl@usehooks{afterextras}{}%
1850   \ifcase\bbl@hymapopt\or
1851     \renewcommand\BabelLower[2]{\lccode##1=##2\relax}%
1852     \ifnum\bbl@hymapsel>4\else
1853       \csname\language @bbl@hyphenmap\endcsname
1854     \fi
1855     \chardef\bbl@hymapopt\z@
1856   \else
1857     \ifnum\bbl@hymapsel>\bbl@hymapopt\else
1858       \csname\language @bbl@hyphenmap\endcsname
1859     \fi
1860   \fi
1861   \global\let\bbl@hymapsel\@cclv
1862   \bbl@patterns{#1}%
1863   \babel@savevariable\lefthyphenmin
1864   \babel@savevariable\righthyphenmin

```



```

1865 \expandafter\ifx\csname #1hyphenmins\endcsname\relax
1866   \set@hyphenmins\tw@\thr@@\relax
1867 \else
1868   \expandafter\expandafter\expandafter\set@hyphenmins
1869   \csname #1hyphenmins\endcsname\relax
1870 \fi}

1871 \def\bbl@ifsamestring#1#2{%
1872   \protected@edef\bbl@tempb{#1}%
1873   \edef\bbl@tempb{\expandafter\strip@prefix\meaning\bbl@tempb}%
1874   \protected@edef\bbl@tempc{#2}%
1875   \edef\bbl@tempc{\expandafter\strip@prefix\meaning\bbl@tempc}%
1876   \ifx\bbl@tempb\bbl@tempc
1877     \expandafter\@firstoftwo
1878   \else
1879     \expandafter\@secondoftwo
1880 \fi}

```

`otherlanguage` The `otherlanguage` environment can be used as an alternative to using the `\selectlanguage` declarative command. When you are typesetting a document which mixes left-to-right and right-to-left typesetting you have to use this environment in order to let things work as you expect them to. The first thing this environment does is store the name of the language in `\language`; it then calls `\selectlanguage_` to switch on everything that is needed for this language. The `\ignorespaces` command is necessary to hide the environment when it is entered in horizontal mode.

```

1881 \long\def\otherlanguage#1{%
1882   \ifnum\bbl@hymapsel=\@cclv\let\bbl@hymapsel\thr@@\fi
1883   \csname selectlanguage \endcsname{#1}%
1884   \ignorespaces}

```

The `\endotherlanguage` part of the environment tries to hide itself when it is called in horizontal mode.

```

1885 \long\def\endotherlanguage{%
1886   \global\@ignoretrue\ignorespaces}

```

`otherlanguage*` The `otherlanguage` environment is meant to be used when a large part of text from a different language needs to be typeset, but without changing the translation of words such as ‘figure’. This environment makes use of `\foreign@language`.

```

1887 \expandafter\def\csname otherlanguage*\endcsname#1{%
1888   \ifnum\bbl@hymapsel=\@cclv\chardef\bbl@hymapsel4\relax\fi
1889   \foreign@language{#1}}

```

At the end of the environment we need to switch off the extra definitions. The grouping mechanism of the environment will take care of resetting the correct hyphenation rules and “extras”.

```

1890 \expandafter\let\csname endotherlanguage*\endcsname\relax

```

`\foreignlanguage` The `\foreignlanguage` command is another substitute for the `\selectlanguage` command. This command takes two arguments, the first argument is the name of the language to use for typesetting the text specified in the second argument. Unlike `\selectlanguage` this command doesn’t switch *everything*, it only switches the hyphenation rules and the extra definitions for the language specified. It does this within a group and assumes the `\extras<lang>` command doesn’t make any `\global` changes. The coding is very similar to part of `\selectlanguage`.

```

1891 \edef\foreignlanguage{%

```

```

1892 \noexpand\protect
1893 \expandafter\noexpand\csname foreignlanguage \endcsname}
1894 \expandafter\def\csname foreignlanguage \endcsname#1#2{%
1895 \begingroup
1896 \foreign@language{#1}%
1897 #2%
1898 \endgroup}

```

`\foreign@language` This macro does the work for `\foreignlanguage` and the other `language*` environment. First we need to store the name of the language and check that it is a known language. Then it just calls `bbl@switch`.

```

1899 \def\foreign@language#1{%
1900 \edef\language#1}%
1901 \bbl@fixname\language
1902 \bbl@iflanguage\language{%
1903 \expandafter\ifx\csname date\language\endcsname\relax
1904 \bbl@warning
1905 {You haven't loaded the language #1\space yet\\%
1906 I'll proceed, but expect unexpected results.\\%
1907 Reported}%
1908 \fi
1909 \let\bbl@select@type@ne
1910 \expandafter\bbl@switch\expandafter{\language}}

```

`\bbl@patterns` This macro selects the hyphenation patterns by changing the `\language` register. If special hyphenation patterns are available specifically for the current font encoding, use them instead of the default. It also sets hyphenation exceptions, but only once, because they are global (here `language \lccode's` has been set, too). `\bbl@hyphenation@` is set to relax until the very first `\babelhyphenation`, so do nothing with this value. If the exceptions for a language (by its number, not its name, so that `:ENC` is taken into account) has been set, then use `\hyphenation` with both global and language exceptions and empty the latter to mark they must not be set again.

```

1911 \let\bbl@hyphlist\empty
1912 \let\bbl@hyphenation@relax
1913 \let\bbl@hymapset=\ccclv
1914 \def\bbl@patterns#1{%
1915 \language=\expandafter\ifx\csname l@#1:\f@encoding\endcsname\relax
1916 \csname l@#1\endcsname
1917 \edef\bbl@tempa{#1}%
1918 \else
1919 \csname l@#1:\f@encoding\endcsname
1920 \edef\bbl@tempa{#1:\f@encoding}%
1921 \fi\relax
1922 \@expandtwoargs\bbl@usehooks{patterns}{#1}{\bbl@tempa}}%
1923 \@ifundefined{bbl@hyphenation@}{%
1924 \begingroup
1925 \@expandtwoargs\in@{, \number\language,}{, \bbl@hyphlist}%
1926 \ifin@else
1927 \@expandtwoargs\bbl@usehooks{hyphenation}{#1}{\bbl@tempa}}%
1928 \hyphenation{%
1929 \bbl@hyphenation@
1930 \@ifundefined{bbl@hyphenation@#1}%
1931 \empty
1932 {\space\csname bbl@hyphenation@#1\endcsname}}%

```

```

1933 \xdef\bbl@hyphlist{\bbl@hyphlist\number\language,}%
1934 \fi
1935 \endgroup}}

```

**hyphenrules** The environment `hyphenrules` can be used to select *just* the hyphenation rules. This environment does *not* change `\language` and when the hyphenation rules specified were not loaded it has no effect. Note however, `\lccode`'s and font encodings are not set at all, so in most cases you should use `otherlanguage*`.

```

1936 \def\hyphenrules#1{%
1937 \edef\language{#1}%
1938 \bbl@fixname\language
1939 \bbl@iflanguage\language{
1940 \expandafter\bbl@patterns\expandafter{\language}%
1941 \languageshorthands{none}%
1942 \expandafter\ifx\csname\language hyphenmins\endcsname\relax
1943 \set@hyphenmins\tw@\thr@@\relax
1944 \else
1945 \expandafter\expandafter\expandafter\set@hyphenmins
1946 \csname\language hyphenmins\endcsname\relax
1947 \fi}}
1948 \let\endhyphenrules\@empty

```

**\providehyphenmins** The macro `\providehyphenmins` should be used in the language definition files to provide a *default* setting for the hyphenation parameters `\lefthyphenmin` and `\righthyphenmin`. If the macro `\<lang>hyphenmins` is already defined this command has no effect.

```

1949 \def\providehyphenmins#1#2{%
1950 \expandafter\ifx\csname #1hyphenmins\endcsname\relax
1951 \namedef{#1hyphenmins}{#2}%
1952 \fi}

```

**\set@hyphenmins** This macro sets the values of `\lefthyphenmin` and `\righthyphenmin`. It expects two values as its argument.

```

1953 \def\set@hyphenmins#1#2{\lefthyphenmin#1\relax\righthyphenmin#2\relax}

```

As an alternative for `\ProvidesFile` we define `\ProvidesLanguage` here to be used in the language definition files. Depending on the format, ie, on if the former is defined, we use a similar definition or not.

```

1954 \ifx\ProvidesFile\@undefined
1955 \def\ProvidesLanguage#1[#2 #3 #4]{%
1956 \wlog{Language: #1 #4 #3 <#2>}%
1957 }
1958 \else
1959 \def\ProvidesLanguage#1{%
1960 \begingroup
1961 \catcode'\ 10 %
1962 \@makeother\/%
1963 \@ifnextchar[%
1964 {\@provideslanguage{#1}}{\@provideslanguage{#1}[]}}
1965 \def\@provideslanguage#1[#2]{%
1966 \wlog{Language: #1 #2}%
1967 \expandafter\xdef\csname ver@#1.ldf\endcsname{#2}%
1968 \endgroup}
1969 \fi

```

`\LdfInit` This macro is defined in two versions. The first version is to be part of the ‘kernel’ of babel, ie. the part that is loaded in the format; the second version is defined in `babel.def`. The version in the format just checks the category code of the ampersand and then loads `babel.def`.

The category code of the ampersand is restored and the macro calls itself again with the new definition from `babel.def`

```
1970 \def\LdfInit{%
1971   \chardef\atcatcode=\catcode'\@
1972   \catcode'\@=11\relax
1973   \input babel.def\relax
1974   \catcode'\@=\atcatcode \let\atcatcode\relax
1975   \LdfInit}
```

`\originalTeX` The macro `\originalTeX` should be known to  $\TeX$  at this moment. As it has to be expandable we `\let` it to `@empty` instead of `\relax`.

```
1976 \ifx\originalTeX\undefined\let\originalTeX\@empty\fi
```

Because this part of the code can be included in a format, we make sure that the macro which initialises the save mechanism, `\babel@beginsave`, is not considered to be undefined.

```
1977 \ifx\babel@beginsave\undefined\let\babel@beginsave\relax\fi
```

`\@nolanerr` The babel package will signal an error when a documents tries to select a language that hasn’t been defined earlier. When a user selects a language for which no hyphenation patterns were loaded into the format he will be given a warning about that fact. We revert to the patterns for `\language=0` in that case. In most formats that will be (US)english, but it might also be empty.

`\@noopterr` When the package was loaded without options not everything will work as expected. An error message is issued in that case.  
When the format knows about `\PackageError` it must be  $\text{\LaTeX} 2_{\epsilon}$ , so we can safely use its error handling interface. Otherwise we’ll have to ‘keep it simple’.

```
1978 \edef\bbl@nulllanguage{\string\language=0}
1979 \ifx\PackageError\undefined
1980   \def\bbl@error#1#2{%
1981     \begingroup
1982       \newlinechar='^^J
1983       \def\{^^J(babel) }%
1984       \errhelp{#2}\errmessage{\{#1}%
1985     \endgroup}
1986   \def\bbl@warning#1{%
1987     \begingroup
1988       \newlinechar='^^J
1989       \def\{^^J(babel) }%
1990       \message{\{#1}%
1991     \endgroup}
1992   \def\bbl@info#1{%
1993     \begingroup
1994       \newlinechar='^^J
1995       \def\{^^J}%
1996       \wlog{#1}%
1997     \endgroup}
1998 \else
1999   \def\bbl@error#1#2{%
2000     \begingroup
```

```

2001      \def\{\MessageBreak}%
2002      \PackageError{babel}{#1}{#2}%
2003      \endgroup}
2004 \def\bbl@warning#1{%
2005     \begingroup
2006     \def\{\MessageBreak}%
2007     \PackageWarning{babel}{#1}%
2008     \endgroup}
2009 \def\bbl@info#1{%
2010     \begingroup
2011     \def\{\MessageBreak}%
2012     \PackageInfo{babel}{#1}%
2013     \endgroup}
2014 \fi
2015 \def\@nolanerr#1{%
2016     \bbl@error
2017     {You haven't defined the language #1\space yet}%
2018     {Your command will be ignored, type <return> to proceed}}
2019 \def\@nopatterns#1{%
2020     \bbl@warning
2021     {No hyphenation patterns were preloaded for\%
2022     the language '#1' into the format.\%
2023     Please, configure your TeX system to add them and\%
2024     rebuild the format. Now I will use the patterns\%
2025     preloaded for \bbl@nulllanguage\space instead}}
2026 \let\bbl@usehooks\@gobbletwo
2027 </kernel>

```

The following code is meant to be read by  $\text{iniTeX}$  because it should instruct  $\text{TeX}$  to read hyphenation patterns. To this end the `docstrip` option `patterns` can be used to include this code in the file `hyphen.cfg`. Code is written with lower level macros.

`toks8` stores info to be shown when the program is run.

We want to add a message to the message  $\text{L}^{\text{A}}\text{TeX}$  2.09 puts in the `\everyjob` register. This could be done by the following code:

```

%   \let\orgeveryjob\everyjob
%   \def\everyjob#1{%
%       \orgeveryjob{#1}%
%       \orgeveryjob\expandafter{\the\orgeveryjob\immediate\write16{%
%           hyphenation patterns for \the\loaded@patterns loaded.}}%
%       \let\everyjob\orgeveryjob\let\orgeveryjob\@undefined}
%

```

The code above redefines the control sequence `\everyjob` in order to be able to add something to the current contents of the register. This is necessary because the processing of hyphenation patterns happens long before  $\text{L}^{\text{A}}\text{TeX}$  fills the register. There are some problems with this approach though.

- When someone wants to use several hyphenation patterns with  $\text{SL}^{\text{A}}\text{TeX}$  the above scheme won't work. The reason is that  $\text{SL}^{\text{A}}\text{TeX}$  overwrites the contents of the `\everyjob` register with its own message.
- Plain  $\text{TeX}$  does not use the `\everyjob` register so the message would not be displayed.

To circumvent this a ‘dirty trick’ can be used. As this code is only processed when creating a new format file there is one command that is sure to be used, `\dump`. Therefore the original `\dump` is saved in `\org@dump` and a new definition is supplied.

To make sure that L<sup>A</sup>T<sub>E</sub>X 2.09 executes the `\@begindocumenthook` we would want to alter `\begin{document}`, but as this done too often already, we add the new code at the front of `\@preamblecmds`. But we can only do that after it has been defined, so we add this piece of code to `\dump`.

This new definition starts by adding an instruction to write a message on the terminal and in the transcript file to inform the user of the preloaded hyphenation patterns.

Then everything is restored to the old situation and the format is dumped.

```

2028 (*patterns)
2029 <<Make sure ProvidesFile is defined>>
2030 \ProvidesFile{hyphen.cfg}[<<date>>] <<version>> Babel hyphens]
2031 \xdef\bbl@format{\jobname}
2032 \ifx\AtBeginDocument\@undefined
2033   \def\@empty{}
2034   \let\orig@dump\dump
2035   \def\dump{%
2036     \ifx\@ztryfc\@undefined
2037       \else
2038         \toks0=\expandafter{\@preamblecmds}%
2039         \edef\@preamblecmds{\noexpand\@begindocumenthook\the\toks0}%
2040         \def\@begindocumenthook{}%
2041       \fi
2042       \let\dump\orig@dump\let\orig@dump\@undefined\dump}
2043 \fi
2044 <<Define core switching macros>>
2045 \toks8{Babel «@version@» and hyphenation patterns for }%
```

`\process@line` Each line in the file `language.dat` is processed by `\process@line` after it is read. The first thing this macro does is to check whether the line starts with `=`. When the first token of a line is an `=`, the macro `\process@synonym` is called; otherwise the macro `\process@language` will continue.

```

2046 \def\process@line#1#2 #3 #4 {%
2047   \ifx=#1%
2048     \process@synonym{#2}%
2049   \else
2050     \process@language{#1#2}{#3}{#4}%
2051   \fi
2052   \ignorespaces}
```

`\process@synonym` This macro takes care of the lines which start with an `=`. It needs an empty token register to begin with. `\bbl@languages` is also set to empty.

```

2053 \toks@{}
2054 \def\bbl@languages{}
```

When no languages have been loaded yet, the name following the `=` will be a synonym for hyphenation register 0. So, it is stored in a token register and executed when the first pattern file has been processed. (The `\relax` just helps to the `\if` below catching synonyms without a language.)

Otherwise the name will be a synonym for the language loaded last.

We also need to copy the `hyphenmin` parameters for the synonym.

```

2055 \def\process@synonym#1{%
2056   \ifnum\last@language=\m@ne
2057     \toks@\expandafter{\the\toks@\relax\process@synonym{#1}}%
2058   \else
2059     \expandafter\chardef\csname l@#1\endcsname\last@language
2060     \wlog{\string\l@#1=\string\language\the\last@language}%
2061     \expandafter\let\csname #1hyphenmins\expandafter\endcsname
2062       \csname\language\hyphenmins\endcsname
2063     \let\bbl@elt\relax
2064     \edef\bbl@languages{\bbl@languages\bbl@elt{#1}{\the\last@language}{}}}%
2065   \fi}

```

`\process@language` The macro `\process@language` is used to process a non-empty line from the ‘configuration file’. It has three arguments, each delimited by white space. The first argument is the ‘name’ of a language; the second is the name of the file that contains the patterns. The optional third argument is the name of a file containing hyphenation exceptions.

The first thing to do is call `\addlanguage` to allocate a pattern register and to make that register ‘active’. Then the ‘name’ of the language that will be loaded now is added to the token register `\toks8`. and finally the pattern file is read. For some hyphenation patterns it is needed to load them with a specific font encoding selected. This can be specified in the file `language.dat` by adding for instance ‘:T1’ to the name of the language. The macro `\bbl@get@enc` extracts the font encoding from the language name and stores it in `\bbl@hyph@enc`. The latter can be used in hyphenation files if you need to set a behaviour depending on the given encoding (it is set to empty if no encoding is given).

Pattern files may contain assignments to `\lefthyphenmin` and `\righthyphenmin`.  $\TeX$  does not keep track of these assignments. Therefore we try to detect such assignments and store them in the `\(lang)hyphenmins` macro. When no assignments were made we provide a default setting.

Some pattern files contain changes to the `\lccode` en `\uccode` arrays. Such changes should remain local to the language; therefore we process the pattern file in a group; the `\patterns` command acts globally so its effect will be remembered. Then we globally store the settings of `\lefthyphenmin` and `\righthyphenmin` and close the group.

When the hyphenation patterns have been processed we need to see if a file with hyphenation exceptions needs to be read. This is the case when the third argument is not empty and when it does not contain a space token. (Note however there is no need to save hyphenation exceptions into the format.)

`\bbl@languages` saves a snapshot of the loaded languages in the form `\bbl@elt{<language-name>}{<number>}{<patterns-file>}{<exceptions-file>}`. Note the last 2 arguments are empty in ‘dialects’ defined in `language.dat` with `=`. Note also the language name can have encoding info.

Finally, if the counter `\language` is equal to zero we execute the synonyms stored.

```

2066 \def\process@language#1#2#3{%
2067   \expandafter\addlanguage\csname l@#1\endcsname
2068   \expandafter\language\csname l@#1\endcsname
2069   \edef\language{#1}%
2070   \bbl@hook@everylanguage{#1}%
2071   \bbl@get@enc#1::\@@@
2072   \begingroup
2073     \lefthyphenmin\m@ne
2074     \bbl@hook@loadpatterns{#2}%

```

```

2075 \ifnum\lefthyphenmin=\m@ne
2076 \else
2077 \expandafter\xdef\csname #1hyphenmins\endcsname{%
2078 \the\lefthyphenmin\the\righthyphenmin}%
2079 \fi
2080 \endgroup
2081 \def\bbl@tempa{#3}%
2082 \ifx\bbl@tempa\@empty\else
2083 \bbl@hook@loadexceptions{#3}%
2084 \fi
2085 \let\bbl@elt\relax
2086 \edef\bbl@languages{%
2087 \bbl@languages\bbl@elt{#1}{\the\language}{#2}{\bbl@tempa}}%
2088 \ifnum\the\language=\z@
2089 \expandafter\ifx\csname #1hyphenmins\endcsname\relax
2090 \set@hyphenmins\tw@\thr@@\relax
2091 \else
2092 \expandafter\expandafter\expandafter\set@hyphenmins
2093 \csname #1hyphenmins\endcsname
2094 \fi
2095 \the\toks@
2096 \toks@{}}%
2097 \fi}

```

`\bbl@get@enc` The macro `\bbl@get@enc` extracts the font encoding from the language name and `\bbl@hyph@enc` stores it in `\bbl@hyph@enc`. It uses delimited arguments to achieve this.

```

2098 \def\bbl@get@enc#1:#2:#3\@@{\def\bbl@hyph@enc{#2}}

```

Now, hooks are defined. For efficiency reasons, they are dealt here in a special way. Besides `luatex`, format specific configuration files are taken into account.

```

2099 \def\bbl@hook@everylanguage#1{}
2100 \def\bbl@hook@loadpatterns#1{\input #1\relax}
2101 \let\bbl@hook@loadexceptions\bbl@hook@loadpatterns
2102 \let\bbl@hook@loadkernel\bbl@hook@loadpatterns
2103 \begingroup
2104 \def\AddBabelHook#1#2{%
2105 \expandafter\ifx\csname bbl@hook@#2\endcsname\relax
2106 \def\next{\toks1}%
2107 \else
2108 \def\next{\expandafter\gdef\csname bbl@hook@#2\endcsname####1}%
2109 \fi
2110 \next}
2111 \ifx\directlua\@undefined
2112 \ifx\XeTeXinputencoding\@undefined\else
2113 \input xebabel.def
2114 \fi
2115 \else
2116 \input luababel.def
2117 \fi
2118 \openin1 = babel-\bbl@format.cfg
2119 \ifeof1
2120 \else
2121 \input babel-\bbl@format.cfg\relax
2122 \fi
2123 \closein1
2124 \endgroup

```



```
2125 \bbl@hook@loadkernel{switch.def}
```

`\readconfigfile` The configuration file can now be opened for reading.

```
2126 \openin1 = language.dat
```

See if the file exists, if not, use the default hyphenation file `hyphen.tex`. The user will be informed about this.

```
2127 \def\language{english}%
2128 \ifeof1
2129   \message{I couldn't find the file language.dat,\space
2130           I will try the file hyphen.tex}
2131   \input hyphen.tex\relax
2132   \chardef\l@english\z@
2133 \else
```

Pattern registers are allocated using count register `\last@language`. Its initial value is 0. The definition of the macro `\newlanguage` is such that it first increments the count register and then defines the language. In order to have the first patterns loaded in pattern register number 0 we initialize `\last@language` with the value `-1`.

```
2134 \last@language\m@ne
```

We now read lines from the file until the end is found

```
2135 \loop
```

While reading from the input, it is useful to switch off recognition of the end-of-line character. This saves us stripping off spaces from the contents of the control sequence.

```
2136 \endlinechar\m@ne
2137 \read1 to \bbl@line
2138 \endlinechar'\^^M
```

If the file has reached its end, exit from the loop here. If not, empty lines are skipped. Add 3 space characters to the end of `\bbl@line`. This is needed to be able to recognize the arguments of `\process@line` later on. The default language should be the very first one.

```
2139 \if T\ifeof1F\fi T\relax
2140 \ifx\bbl@line\@empty\else
2141   \edef\bbl@line{\bbl@line\space\space\space}%
2142   \expandafter\process@line\bbl@line\relax
2143 \fi
2144 \repeat
```

Check for the end of the file. We must reverse the test for `\ifeof` without `\else`. Then reactivate the default patterns,

```
2145 \begingroup
2146 \def\bbl@elt#1#2#3#4{%
2147   \global\language=#2\relax
2148   \gdef\language{#1}%
2149   \def\bbl@elt##1##2##3##4{}}%
2150 \bbl@languages
2151 \endgroup
2152 \fi
```

and close the configuration file.

```
2153 \closein1
```

We add a message about the fact that babel is loaded in the format and with which language patterns to the `\everyjob` register.

```

2154 \if/\the\toks@/\else
2155   \errhelp{language.dat loads no language, only synonyms}
2156   \errmessage{Orphan language synonym}
2157 \fi
2158 \edef\bbl@tempa{%
2159   \everyjob{%
2160     \the\everyjob
2161     \ifx\typeout\@undefined
2162       \immediate\writel6%
2163     \else
2164       \noexpand\typeout
2165     \fi
2166     {\the\toks8 \the\last@language\space languages loaded.}}
2167 \bbl@tempa

```

Also remove some macros from memory and raise an error if `\toks@` is not empty. Finally load `switch.def`, but the letter is not required and the line inputting it may be commented out.

```

2168 \let\bbl@line\@undefined
2169 \let\process@line\@undefined
2170 \let\process@synonym\@undefined
2171 \let\process@language\@undefined
2172 \let\bbl@get@enc\@undefined
2173 \let\bbl@hyph@enc\@undefined
2174 \let\bbl@tempa\@undefined
2175 \let\bbl@hook@loadkernel\@undefined
2176 \let\bbl@hook@everylanguage\@undefined
2177 \let\bbl@hook@loadpatterns\@undefined
2178 \let\bbl@hook@loadexceptions\@undefined
2179 \</patterns>

```

Here the code for `iniTEX` ends.

## 8 The ‘nil’ language

This ‘language’ does nothing, except setting the hyphenation patterns to `nohyphenation`.

For this language currently no special definitions are needed or available.

The macro `\LdfInit` takes care of preventing that this file is loaded more than once, checking the category code of the `@` sign, etc.

```

2180 < *nil>
2181 \ProvidesLanguage{nil}[<<date>> <<version>> Nil language]
2182 \LdfInit{nil}{datenil}

```

When this file is read as an option, i.e. by the `\usepackage` command, `nil` could be an ‘unknown’ language in which case we have to make it known.

```

2183 \ifx\l@nohyphenation\@undefined
2184   \@nopatterns{nil}
2185   \adddialect\l@nil0
2186 \else
2187   \let\l@nil\l@nohyphenation
2188 \fi

```

This macro is used to store the values of the hyphenation parameters `\lefthyphenmin` and `\righthyphenmin`.

```
2189 \providehyphenmins{\CurrentOption}{\m@ne\m@ne}
```

The next step consists of defining commands to switch to (and from) the ‘nil’ language.

```
\captionnil
```

```
\datenil 2190 \let\captionnil\@empty
```

```
2191 \let\datenil\@empty
```

The macro `\ldf@finish` takes care of looking for a configuration file, setting the main language to be switched on at `\begin{document}` and resetting the category code of `@` to its original value.

```
2192 \ldf@finish{nil}
```

```
2193 \end{nil}
```

## 9 Support for Plain T<sub>E</sub>X

### 9.1 Not renaming `hyphen.tex`

As Don Knuth has declared that the filename `hyphen.tex` may only be used to designate *his* version of the american English hyphenation patterns, a new solution has to be found in order to be able to load hyphenation patterns for other languages in a plain-based T<sub>E</sub>X-format. When asked he responded:

That file name is “sacred”, and if anybody changes it they will cause severe upward/downward compatibility headaches.

People can have a file `locallyhyphen.tex` or whatever they like, but they mustn’t diddle with `hyphen.tex` (or `plain.tex` except to preload additional fonts).

The files `bplain.tex` and `lplain.tex` can be used as replacement wrappers around `plain.tex` and `lplain.tex` to achieve the desired effect, based on the `babel` package. If you load each of them with `iniTEX`, you will get a file called either `bplain.fmt` or `lplain.fmt`, which you can use as replacements for `plain.fmt` and `lplain.fmt`.

As these files are going to be read as the first thing `iniTEX` sees, we need to set some category codes just to be able to change the definition of `\input`

```
2194 (*bplain | lplain)
```

```
2195 \catcode'\{=1 % left brace is begin-group character
```

```
2196 \catcode'\}=2 % right brace is end-group character
```

```
2197 \catcode'\#=6 % hash mark is macro parameter character
```

Now let’s see if a file called `hyphen.cfg` can be found somewhere on T<sub>E</sub>X’s input path by trying to open it for reading...

```
2198 \openin 0 hyphen.cfg
```

If the file wasn’t found the following test turns out true.

```
2199 \ifeof0
```

```
2200 \else
```

When `hyphen.cfg` could be opened we make sure that *it* will be read instead of the file `hyphen.tex` which should (according to Don Knuth’s ruling) contain the american English hyphenation patterns and nothing else.

We do this by first saving the original meaning of `\input` (and I use a one letter control sequence for that so as not to waste multi-letter control sequence on this in the format).

```
2201 \let\input
```

Then `\input` is defined to forget about its argument and load `hyphen.cfg` instead.

```
2202 \def\input #1 {%
2203   \let\input\input
2204   \input{hyphen.cfg}
```

Once that's done the original meaning of `\input` can be restored and the definition of `\input` can be forgotten.

```
2205 \let\input\undefined
2206 }
2207 \fi
2208 \bblatex\blatex
```

Now that we have made sure that `hyphen.cfg` will be loaded at the right moment it is time to load `plain.tex`.

```
2209 \bblatex\blatex
2210 \bblatex\blatex
```

Finally we change the contents of `\fmtname` to indicate that this is *not* the plain format, but a format based on plain with the `babel` package preloaded.

```
2211 \bblatex\def\fmtname{babel-plain}
2212 \bblatex\def\fmtname{babel-lplain}
```

When you are using a different format, based on `plain.tex` you can make a copy of `blatex.tex`, rename it and replace `plain.tex` with the name of your format file.

## 9.2 Emulating some L<sup>A</sup>T<sub>E</sub>X features

The following code duplicates or emulates parts of L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> that are needed for `babel`.

We need to define `\loadlocalcfg` for plain users as the L<sup>A</sup>T<sub>E</sub>X definition uses `\InputIfFileExists`. We have to execute `\@endofldef` in this case.

```
2213 \bblatex
2214 \def\@empty{}
2215 \def\loadlocalcfg#1{%
2216   \openin0#1.cfg
2217   \ifeof0
2218     \closein0
2219   \else
2220     \closein0
2221     {\immediate\write16{*****}%
2222      \immediate\write16{* Local config file #1.cfg used}%
2223      \immediate\write16{**}%
2224     }
2225     \input #1.cfg\relax
2226   \fi
2227   \@endofldef}
```

A number of L<sup>A</sup>T<sub>E</sub>X macro's that are needed later on.

```
2228 \long\def\@firstofone#1{#1}
2229 \long\def\@firstoftwo#1#2{#1}
2230 \long\def\@secondoftwo#1#2{#2}
2231 \def\@ifstar#1{\@ifnextchar *{\@firstoftwo{#1}}}
```

```

2232 \def\@star@or@long#1{%
2233   \@ifstar
2234   {\let\l@ngrel@x\relax#1}%
2235   {\let\l@ngrel@x\long#1}}
2236 \let\l@ngrel@x\relax
2237 \def\@car#1#2\@nil{#1}
2238 \def\@cdr#1#2\@nil{#2}
2239 \let\@typeset@protect\relax
2240 \long\def\@gobble#1{}
2241 \edef\@backslashchar{\expandafter\@gobble\string\}
2242 \def\strip@prefix#1>{}
2243 \def\g@addto@macro#1#2{%
2244   \toks@\expandafter{#1#2}%
2245   \xdef#1{\the\toks@}}
2246 \def\@namedef#1{\expandafter\def\csname #1\endcsname}
2247 \def\@nameuse#1{\csname #1\endcsname}
2248 \def\@ifundefined#1{%
2249   \expandafter\ifx\csname#1\endcsname\relax
2250     \expandafter\@firstoftwo
2251   \else
2252     \expandafter\@secondoftwo
2253   \fi}
2254 \def\@expandtwoargs#1#2#3{%
2255   \edef\reserved@a{\noexpand#1{#2}{#3}}\reserved@a}

```

L<sup>A</sup>T<sub>E</sub>X<sub>2<sub>ε</sub></sub> has the command \@onlypreamble which adds commands to a list of commands that are no longer needed after \begin{document}.

```

2256 \ifx\@preamblecmds\@undefined
2257   \def\@preamblecmds{}
2258 \fi
2259 \def\@onlypreamble#1{%
2260   \expandafter\gdef\expandafter\@preamblecmds\expandafter{
2261     \@preamblecmds\do#1}}
2262 \@onlypreamble\@onlypreamble

```

Mimick L<sup>A</sup>T<sub>E</sub>X's \AtBeginDocument; for this to work the user needs to add \begindocument to his file.

```

2263 \def\begindocument{%
2264   \@begindocumenthook
2265   \global\let\@begindocumenthook\@undefined
2266   \def\do##1{\global\let ##1\@undefined}%
2267   \@preamblecmds
2268   \global\let\do\noexpand
2269   }
2270 \ifx\@begindocumenthook\@undefined
2271   \def\@begindocumenthook{}
2272 \fi
2273 \@onlypreamble\@begindocumenthook
2274 \def\AtBeginDocument{\g@addto@macro\@begindocumenthook}

```

We also have to mimick L<sup>A</sup>T<sub>E</sub>X's \AtEndOfPackage. Our replacement macro is much simpler; it stores its argument in \@endofldf.

```

2275 \def\AtEndOfPackage#1{\g@addto@macro\@endofldf{#1}}
2276 \@onlypreamble\AtEndOfPackage
2277 \def\@endofldf{}
2278 \@onlypreamble\@endofldf
2279 \let\bb@afterlang\@empty

```

L<sup>A</sup>T<sub>E</sub>X needs to be able to switch off writing to its auxiliary files; plain doesn't have them by default.

```
2280 \ifx\if@filesw\undefined
2281   \expandafter\let\csname if@filesw\expandafter\endcsname
2282     \csname iffalse\endcsname
2283 \fi
```

Mimick L<sup>A</sup>T<sub>E</sub>X's commands to define control sequences.

```
2284 \def\newcommand{\@star@or@long\new@command}
2285 \def\new@command#1{%
2286   \@testopt{\@newcommand#1}0}
2287 \def\@newcommand#1[#2]{%
2288   \@ifnextchar [{\@xargdef#1[#2]}%
2289                 {\@argdef#1[#2]}}
2290 \long\def\@argdef#1[#2]#3{%
2291   \@yargdef#1\@ne{#2}{#3}}
2292 \long\def\@xargdef#1[#2][#3]#4{%
2293   \expandafter\def\expandafter#1\expandafter{%
2294     \expandafter\@protected@testopt\expandafter #1%
2295     \csname\string#1\expandafter\endcsname{#3}}}%
2296   \expandafter\@yargdef \csname\string#1\endcsname
2297   \tw@{#2}{#4}}
2298 \long\def\@yargdef#1#2#3{%
2299   \@tempcnta#3\relax
2300   \advance \@tempcnta \@ne
2301   \let\@hash@\relax
2302   \edef\reserved@a{\ifx#2\tw@ [\@hash@1]\fi}%
2303   \@tempcntb #2%
2304   \@whilenum\@tempcntb <\@tempcnta
2305   \do{%
2306     \edef\reserved@a{\reserved@a\@hash@\the\@tempcntb}%
2307     \advance\@tempcntb \@ne}%
2308   \let\@hash@##%
2309   \l@ngrel@x\expandafter\def\expandafter#1\reserved@a}
2310 \let\providecommand\newcommand

2311 \def\DeclareRobustCommand{\@star@or@long\declare@robustcommand}
2312 \def\declare@robustcommand#1{%
2313   \edef\reserved@a{\string#1}%
2314   \def\reserved@b{#1}%
2315   \edef\reserved@b{\expandafter\strip@prefix\meaning\reserved@b}%
2316   \edef#1{%
2317     \ifx\reserved@a\reserved@b
2318       \noexpand\x@protect
2319       \noexpand#1%
2320     \fi
2321     \noexpand\protect
2322     \expandafter\new@command\csname
2323       \expandafter\@gobble\string#1 \endcsname
2324   }%
2325   \expandafter\new@command\csname
2326     \expandafter\@gobble\string#1 \endcsname
2327 }
2328 \def\x@protect#1{%
2329   \ifx\protect\@typeset@protect\else
2330     \@x@protect#1%
2331   \fi
```

```

2332 }
2333 \def\@x@protect#1\fi#2#3{%
2334   \fi\protect#1%
2335 }

```

The following little macro `\in@` is taken from `latex.ltx`; it checks whether its first argument is part of its second argument. It uses the boolean `\in@`; allocating a new boolean inside conditionally executed code is not possible, hence the construct with the temporary definition of `\bbl@tempa`.

```

2336 \def\bbl@tempa{\csname newif\endcsname\ifin@}
2337 \ifx\in@\@undefined
2338   \def\in@#1#2{%
2339     \def\in@@##1#1##2##3\in@@{%
2340       \ifx\in@@##2\in@false\else\in@true\fi}%
2341     \in@@#2#1\in@\in@@}
2342 \else
2343   \let\bbl@tempa\@empty
2344 \fi
2345 \bbl@tempa

```

$\text{\LaTeX}$  has a macro to check whether a certain package was loaded with specific options. The command has two extra arguments which are code to be executed in either the true or false case. This is used to detect whether the document needs one of the accents to be activated (`activegrave` and `activeacute`). For plain  $\text{\TeX}$  we assume that the user wants them to be active by default. Therefore the only thing we do is execute the third argument (the code for the true case).

```

2346 \def\@ifpackagewith#1#2#3#4{#3}

```

The  $\text{\LaTeX}$  macro `\@ifl@aded` checks whether a file was loaded. This functionality is not needed for plain  $\text{\TeX}$  but we need the macro to be defined as a no-op.

```

2347 \def\@ifl@aded#1#2#3#4{}

```

For the following code we need to make sure that the commands `\newcommand` and `\providecommand` exist with some sensible definition. They are not fully equivalent to their  $\text{\LaTeX}2_{\epsilon}$  versions; just enough to make things work in plain  $\text{\TeX}$  environments.

```

2348 \ifx\@tempcnta\@undefined
2349   \csname newcount\endcsname\@tempcnta\relax
2350 \fi
2351 \ifx\@tempcntb\@undefined
2352   \csname newcount\endcsname\@tempcntb\relax
2353 \fi

```

To prevent wasting two counters in  $\text{\LaTeX} 2.09$  (because counters with the same name are allocated later by it) we reset the counter that holds the next free counter (`\count10`).

```

2354 \ifx\bye\@undefined
2355   \advance\count10 by -2\relax
2356 \fi
2357 \ifx\@ifnextchar\@undefined
2358   \def\@ifnextchar#1#2#3{%
2359     \let\reserved@d=#1%
2360     \def\reserved@a{#2}\def\reserved@b{#3}%
2361     \futurelet\@let@token\@ifnch}
2362   \def\@ifnch{%
2363     \ifx\@let@token\@sptoken

```

```

2364     \let\reserved@c\@xifnch
2365   \else
2366     \ifx\@let@token\reserved@d
2367       \let\reserved@c\reserved@a
2368     \else
2369       \let\reserved@c\reserved@b
2370     \fi
2371   \fi
2372   \reserved@c}
2373 \def\:{\let\@sptoken= } \: % this makes \@sptoken a space token
2374 \def\:{\@xifnch} \expandafter\def\:{\futurelet\@let@token\@ifnch}
2375 \fi
2376 \def\@testopt#1#2{%
2377   \ifnextchar[{\#1}{\#1[\#2]}}
2378 \def\@protected@testopt#1{%
2379   \ifx\protect\@typeset@protect
2380     \expandafter\@testopt
2381   \else
2382     \@x@protect#1%
2383   \fi}
2384 \long\def\@whilenum#1\do #2{\ifnum #1\relax #2\relax\@iwhilenum{#1\relax
2385   #2\relax}\fi}
2386 \long\def\@iwhilenum#1{\ifnum #1\expandafter\@iwhilenum
2387   \else\expandafter\@gobble\fi{#1}}

```

Code from `ltoutenc.dtx`, adapted for use in the plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  environment.

```

2388 \def\DeclareTextCommand{%
2389   \@dec@text@cmd\providecommand
2390 }
2391 \def\ProvideTextCommand{%
2392   \@dec@text@cmd\providecommand
2393 }
2394 \def\DeclareTextSymbol#1#2#3{%
2395   \@dec@text@cmd\chardef#1{#2}#3\relax
2396 }
2397 \def\@dec@text@cmd#1#2#3{%
2398   \expandafter\def\expandafter#2%
2399     \expandafter{%
2400       \csname#3-cmd\expandafter\endcsname
2401       \expandafter#2%
2402       \csname#3\string#2\endcsname
2403     }%
2404 %   \let\@ifdefinable\@rc@ifdefinable
2405   \expandafter#1\csname#3\string#2\endcsname
2406 }
2407 \def\@current@cmd#1{%
2408   \ifx\protect\@typeset@protect\else
2409     \noexpand#1\expandafter\@gobble
2410   \fi
2411 }
2412 \def\@changed@cmd#1#2{%
2413   \ifx\protect\@typeset@protect
2414     \expandafter\ifx\csname\cf@encoding\string#1\endcsname\relax
2415       \expandafter\ifx\csname ?\string#1\endcsname\relax
2416         \expandafter\def\csname ?\string#1\endcsname{%
2417           \@changed@x@err{#1}%

```



```

2418         }%
2419     \fi
2420     \global\expandafter\let
2421         \csname\cf@encoding \string#1\expandafter\endcsname
2422         \csname ?\string#1\endcsname
2423     \fi
2424     \csname\cf@encoding\string#1%
2425         \expandafter\endcsname
2426 \else
2427     \noexpand#1%
2428 \fi
2429 }
2430 \def\@changed@x@err#1{%
2431     \errhelp{Your command will be ignored, type <return> to proceed}%
2432     \errmessage{Command \protect#1 undefined in encoding \cf@encoding}}
2433 \def\DeclareTextCommandDefault#1{%
2434     \DeclareTextCommand#1?%
2435 }
2436 \def\ProvideTextCommandDefault#1{%
2437     \ProvideTextCommand#1?%
2438 }
2439 \expandafter\let\csname OT1-cmd\endcsname\@current@cmd
2440 \expandafter\let\csname?-cmd\endcsname\@changed@cmd
2441 \def\DeclareTextAccent#1#2#3{%
2442     \DeclareTextCommand#1{#2}[1]{\accent#3 ##1}
2443 }
2444 \def\DeclareTextCompositeCommand#1#2#3#4{%
2445     \expandafter\let\expandafter\reserved@a\csname#2\string#1\endcsname
2446     \edef\reserved@b{\string##1}%
2447     \edef\reserved@c{%
2448         \expandafter\@strip@args\meaning\reserved@a:-\@strip@args}%
2449     \ifx\reserved@b\reserved@c
2450         \expandafter\expandafter\expandafter\ifx
2451             \expandafter\@car\reserved@a\relax\relax\@nil
2452             \@text@composite
2453     \else
2454         \edef\reserved@b##1{%
2455             \def\expandafter\noexpand
2456                 \csname#2\string#1\endcsname####1{%
2457                 \noexpand\@text@composite
2458                     \expandafter\noexpand\csname#2\string#1\endcsname
2459                     ####1\noexpand\@empty\noexpand\@text@composite
2460                     {##1}%
2461             }%
2462         }%
2463         \expandafter\reserved@b\expandafter{\reserved@a{##1}}%
2464     \fi
2465     \expandafter\def\csname\expandafter\string\csname
2466         #2\endcsname\string#1-\string#3\endcsname{#4}
2467 \else
2468     \errhelp{Your command will be ignored, type <return> to proceed}%
2469     \errmessage{\string\DeclareTextCompositeCommand\space used on
2470         inappropriate command \protect#1}
2471 \fi
2472 }
2473 \def\@text@composite#1#2#3\@text@composite{%

```

```

2474 \expandafter\@text@composite@x
2475 \csname\string#1-\string#2\endcsname
2476 }
2477 \def\@text@composite@x#1#2{%
2478 \ifx#1\relax
2479 #2%
2480 \else
2481 #1%
2482 \fi
2483 }
2484 %
2485 \def\@strip@args#1:#2-#3\@strip@args{#2}
2486 \def\DeclareTextComposite#1#2#3#4{%
2487 \def\reserved@a{\DeclareTextCompositeCommand#1{#2}{#3}}%
2488 \bgroup
2489 \lccode'\@=#4%
2490 \lowercase{%
2491 \egroup
2492 \reserved@a @%
2493 }%
2494 }
2495 %
2496 \def\UseTextSymbol#1#2{%
2497 % \let\@curr@enc\cf@encoding
2498 % \@use@text@encoding{#1}%
2499 #2%
2500 % \@use@text@encoding\@curr@enc
2501 }
2502 \def\UseTextAccent#1#2#3{%
2503 % \let\@curr@enc\cf@encoding
2504 % \@use@text@encoding{#1}%
2505 % #2{\@use@text@encoding\@curr@enc\selectfont#3}%
2506 % \@use@text@encoding\@curr@enc
2507 }
2508 \def\@use@text@encoding#1{%
2509 % \edef\@f@encoding{#1}%
2510 % \xdef\font@name{%
2511 % \csname\curr@fontshape/\@f@size\endcsname
2512 % }%
2513 % \pickup@font
2514 % \font@name
2515 % @@enc@update
2516 }
2517 \def\DeclareTextSymbolDefault#1#2{%
2518 \DeclareTextCommandDefault#1{\UseTextSymbol{#2}#1}%
2519 }
2520 \def\DeclareTextAccentDefault#1#2{%
2521 \DeclareTextCommandDefault#1{\UseTextAccent{#2}#1}%
2522 }
2523 \def\cf@encoding{OT1}

```

Currently we only use the L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> method for accents for those that are known to be made active in *some* language definition file.

```

2524 \DeclareTextAccent{"}{OT1}{127}
2525 \DeclareTextAccent{'}{OT1}{19}
2526 \DeclareTextAccent{^}{OT1}{94}

```

```

2527 \DeclareTextAccent{\'}{OT1}{18}
2528 \DeclareTextAccent{\~}{OT1}{126}

```

The following control sequences are used in `babel.def` but are not defined for plain  $\text{T}_{\text{E}}\text{X}$ .

```

2529 \DeclareTextSymbol{\textquotedblleft}{OT1}{92}
2530 \DeclareTextSymbol{\textquotedblright}{OT1}{'\'}
2531 \DeclareTextSymbol{\textquoteleft}{OT1}{'\'}
2532 \DeclareTextSymbol{\textquoteright}{OT1}{'\'}
2533 \DeclareTextSymbol{\i}{OT1}{16}
2534 \DeclareTextSymbol{\ss}{OT1}{25}

```

For a couple of languages we need the  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ -control sequence `\scriptsize` to be available. Because plain  $\text{T}_{\text{E}}\text{X}$  doesn't have such a sophisticated font mechanism as  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$  has, we just `\let` it to `\sevenrm`.

```

2535 \ifx\scriptsize\@undefined
2536   \let\scriptsize\sevenrm
2537 \fi
2538 \</plain>

```

## 10 Tentative font handling

A general solution is far from trivial:

- `\addfontfeature` only sets it for the current family and it's not very efficient, and
- `\defaultfontfeatures` requires to redefine the font (and the options aren't "orthogonal").

```
«*Font selection» langfeaturesempty «/Font selection»
```

## 11 Hooks for XeTeX and LuaTeX

### 11.1 XeTeX

Unfortunately, the current encoding cannot be retrieved and therefore it is reset always to `utf8`, which seems a sensible default.

```

2539 \<{*xetex}>
2540 \def\BabelStringsDefault{unicode}
2541 \let\xebbl@stop\relax
2542 \AddBabelHook{xetex}{encodedcommands}{%
2543   \def\bbl@tempa{#1}%
2544   \ifx\bbl@tempa\@empty
2545     \XeTeXinputencoding"bytes"%
2546   \else
2547     \XeTeXinputencoding"#1"%
2548   \fi
2549   \def\xebbl@stop{\XeTeXinputencoding"utf8"}}
2550 \AddBabelHook{xetex}{stopcommands}{%
2551   \xebbl@stop
2552   \let\xebbl@stop\relax}
2553 \<<Font selection>>
2554 \</xetex>

```

## 11.2 LuaTeX

This part relies on the lua strips in luatex-hyphen by Khaled Hosny, Élie Roux, and Manuel Pégourié-Gonnard. Élie also improved the code below.

```
2555 <lua>
2556 \directlua{%
2557   require("luatex-hyphen")
2558   Babel = {}
2559   function Babel.bytes(line)
2560     return line:gsub(".",
2561       function (chr) return unicode.utf8.char(string.byte(chr)) end)
2562   end
2563   function Babel.begin_process_input()
2564     if luatexbase and luatexbase.add_to_callback then
2565       luatexbase.add_to_callback('process_input_buffer', Babel.bytes, 'Babel.bytes')
2566     else
2567       Babel.callback = callback.find('process_input_buffer')
2568       callback.register('process_input_buffer', Babel.bytes)
2569     end
2570   end
2571   function Babel.end_process_input ()
2572     if luatexbase and luatexbase.remove_from_callback then
2573       luatexbase.remove_from_callback('process_input_buffer', 'Babel.bytes')
2574     else
2575       callback.register('process_input_buffer', Babel.callback)
2576     end
2577   end
2578 }
2579 \def\BabelStringsDefault{unicode}
2580 \let\luabbl@stop\relax
2581 \AddBabelHook{luatex}{encodedcommands}{%
2582   \def\bbl@tempa{utf8}\def\bbl@tempb{#1}%
2583   \ifx\bbl@tempa\bbl@tempb\else
2584     \directlua{Babel.begin_process_input()}%
2585   \def\luabbl@stop{%
2586     \directlua{Babel.end_process_input()}}%
2587   \fi}%
2588 \AddBabelHook{luatex}{stopcommands}{%
2589   \luabbl@stop
2590   \let\luabbl@stop\relax}
2591 \AddBabelHook{luatex}{patterns}{%
2592   \ifx\directlua\relax\else
2593     \ifcsname lu@texhyphen@loaded@the\language\endcsname \else
2594       \global\@namedef{lu@texhyphen@loaded@the\language}{}%
2595       \directlua{
2596         luatexhyphen.loadlanguage('\luatexluaescapestring{\string#1}',
2597           '\the\language')}%
2598       \fi
2599     \fi}
2600 \AddBabelHook{luatex}{adddialect}{%
2601   \ifx\directlua\relax\else
2602     \directlua{
2603       luatexhyphen.adddialect('\luatexluaescapestring{\string#1}',
2604         '\luatexluaescapestring{\string#2'})
2605     }%
2606   \fi}
```

```

2607 \AddBabelHook{luatex}{everylanguage}{%
2608   \directlua{
2609     processnow = (tex.language == 0) or
2610       (luatexhyphen.lookupname('\luatexluaescapestring{\string#1}') == nil)}%
2611   \ifnum0=\directlua{tex.sprint(processnow and "0" or "1")}\relax
2612   \global\@namedef{lu@texhyphen@loaded@the\language}{}%
2613   \fi}
2614 \AddBabelHook{luatex}{loadpatterns}{%
2615   \ifnum0=\directlua{tex.sprint(processnow and "0" or "1")}\relax
2616   \input #1\relax
2617   \fi}
2618 \AddBabelHook{luatex}{loadexceptions}{%
2619   \ifnum0=\directlua{tex.sprint(processnow and "0" or "1")}\relax
2620   \input #1\relax
2621   \fi
2622   \directlua{processnow = nil}}
2623 <<Font selection>>
2624 </luatex>

```

## 12 Conclusion

A system of document options has been presented that enable the user of L<sup>A</sup>T<sub>E</sub>X to adapt the standard document classes of L<sup>A</sup>T<sub>E</sub>X to the language he or she prefers to use. These options offer the possibility of switching between languages in one document. The basic interface consists of using one option, which is the same for *all* standard document classes.

In some cases the language definition files provide macros that can be useful to plain T<sub>E</sub>X users as well as to L<sup>A</sup>T<sub>E</sub>X users. The babel system has been implemented so that it can be used by both groups of users.

## 13 Acknowledgements

I would like to thank all who volunteered as  $\beta$ -testers for their time. I would like to mention Julio Sanchez who supplied the option file for the Spanish language and Maurizio Codogno who supplied the option file for the Italian language. Michel Goossens supplied contributions for most of the other languages. Nico Poppelier helped polish the text of the documentation and supplied parts of the macros for the Dutch language. Paul Wackers and Werenfried Spit helped find and repair bugs.

During the further development of the babel system I received much help from Bernd Raichle, for which I am grateful.

## References

- [1] Donald E. Knuth, *The T<sub>E</sub>Xbook*, Addison-Wesley, 1986.
- [2] Leslie Lamport, *L<sup>A</sup>T<sub>E</sub>X, A document preparation System*, Addison-Wesley, 1986.
- [3] K.F. Treebus. *Tekstwijzer, een gids voor het grafisch verwerken van tekst*. SDU Uitgeverij ('s-Gravenhage, 1988). A Dutch book on layout design and typography.

- [4] Hubert Partl, *German T<sub>E</sub>X*, *TUGboat* 9 (1988) #1, p. 70–72.
- [5] Leslie Lamport, in: T<sub>E</sub>Xhax Digest, Volume 89, #13, 17 February 1989.
- [6] Johannes Braams, Victor Eijkhout and Nico Poppelier, *The development of national L<sup>A</sup>T<sub>E</sub>X styles*, *TUGboat* 10 (1989) #3, p. 401–406.
- [7] Joachim Schrod, *International L<sup>A</sup>T<sub>E</sub>X is ready to use*, *TUGboat* 11 (1990) #1, p. 87–90.