The fontinst utility

Alan Jeffrey, Sebastian Rahtz, Ulrik Vieth, Lars Hellström

November 16, 1999

Contents

About the fontinst package

This document describes version 1.9 of fontinst. This is a prerelease, and in particular this document (the commented source) has much of an internal debate paper about it. The files generated from the source should however work (there may of course be various bugs in them).

1 Files

1.1 Source files

The source for fontinst is currently split on the five source files `fibasics.dtx`, `fimain.dtx`, `ficonv.dtx`, `filtfam.dtx`, and `fimapgen.dtx`.

1.2 Generated files

By running the docstrip installation script `fontinst.ins`, the following files are generated:

`fontinst.sty` This is the executable file that contains all the commands that set up the fontinst utility for use.

`xfntinst.sty` This is a variant of `fontinst.sty` which behaves differently with respect to from which fonts the digits are picked (see below).

`cfntinst.sty` This is a variant of `fontinst.sty` which supports the old way of selecting boundarychar for fonts (i.e., by setting the integer `boundarychar`). It requires a bit more memory than `fontinst.sty` does.

`fontinst.ini` This file contains some very basic initialization code. It is used in case fontinst is used under iniTEX without a preloaded format.

`finstmsc.sty` This file contains the routines for automatic map file generation, which is still a part of the fontinst system that is under construction, but which does work as of v 1.904. This file also contains some seldom-used code (the ENC to ETX conversion) that has been broken off from the main fontinst utility, as it almost always just sat there and used up space for no good reason.

`fontdoc.sty` This file contains the fontdoc package, which can be used to typeset encoding and metric files.

`csc2x.tex` This file contains the reglyphing commands that set up the common caps-and-small-caps to expert names conversion. See Subsections 3.3 and 16.3 for details.

`csckrn2x.tex` This file is a variant of `csc2x.tex`, which doesn't copy the commands that actually set glyphs.

1.3 Modules

The source is split up on the following docstrip modules:

⟨pkg⟩, **all files** Guards code that is to go into `fontinst.sty` (and its variants).

⟨doc⟩, `fibasics` **and** `fimain` Guards code that is to go into `fontdoc.sty`.

⟨driver⟩, **all files** Guards the driver code—the short piece of LaTeX code in the beginning of each file that makes it possible to typeset as a LaTeX document. `docstrip.ins` doesn't extract this code, but it is possible to write a script that makes docstrip extract this code if one wants to write a modified driver. It's not usual, though.

⟨pkg2⟩, `fibasics` A special variant of ⟨pkg⟩ that is used for code that is to end up at the very end of `fontinst.sty`.

⟨ini⟩, `fibasics` Guards the code for `fontinst.ini`.

⟨everyjob⟩, `fibasics` Guards the code setting `\everyjob` for fontinst formats (by default not used).

⟨misc⟩, `fibasics`, `fimain`, **and** `ficonv` Guards code that is only to go into `finstmsc.sty`.

⟨!misc⟩, `fibasics` Guards code that is not to go into `finstmsc.sty`.

⟨oldTeX⟩, `fibasics` Guards code that copes with a bug in TeX versions before 3.141.

⟨!oldTeX⟩, `fibasics` Guards normal code that wouldn't work satisfactory for TeX versions before 3.141 due to a bug in those TeXs.

⟨boundaryCompability⟩, `fimain` Guards code that is needed for fontinst to be compatible with the old interface for boundary ligatures and kerns.

⟨!boundaryCompability⟩, `fimain` Guards code that is used when fontinst is not to be compatible with the old interface for boundary ligatures and kerns.

⟨obsolete⟩, `fimain` Guards miscellaneous pieces of code that are considered obsolete.

⟨reglyph⟩, `ficonv` Guards code that is to go to the reglyphing setup files `csc2x.tex` and `csckrn2x.tex`.

⟨!glyphs⟩, `ficonv` Guards reglyphing setup code that is relevant when commands that set glyphs are not to be copied.

⟨glyphs⟩, `ficonv` Guards reglyphing setup code that is relevant when commands that set glyphs are to be copied.

⟨textcomp⟩, `filtfam` Guards code in the `\latinfamily` section that generates text symbol (`TS1`/"text companion") fonts.

⟨debug⟩, `filtfam` Guards code in the `\latinfamily` section that writes `INFO>` messages.

⟨oldstyle⟩, `filtfam` Guards code in the `\latinfamily` section. The purpose of that code is described in more detail in Section 4.

⟨map⟩, `fimapgen` Guards the code for the automatic map file generation.

2 About previous fontinst releases

2.1 Pre-doc fontinst releases

The following is a choice of some fontinst versions and some comments about them. The complete list of versions in the range v 0.01–1.7 can be found in the file `CHANGES`.

v 0.01 was begun by Alan Jeffrey in February 1993. This was the very first version recorded in the `CHANGES` file.

v 0.19 was completed in April 1993 and presented at the TUG '93 conference in July 1993. It is described in the proceedings in *TUGboat* 14#3 and represents one of the earliest release versions, in which the user interface was still subject to change.

v 1.0 was begun in August 1993 right after the conference and represents the beginning of a complete re-write from scratch.

v 1.333 was presented one year later at the TUG '94 conference and is described in the proceedings in *TUGboat* 15#3. It uses a different user interface, which is largely the same as in the present version, although the internals of the implementation have subsequently changed quite a bit.

v 1.335 was released in September 1994 and was the last "official" release version by Alan Jeffrey. Until the summer of 1998 this was still the version to be found in the `fonts/utilities/fontinst` directory on CTAN. It is considered obsolete and should not be used any longer.

v 1.400 was prepared in November 1994 and was the first version which used `8r`-encoded raw fonts as a basis for virtual fonts.

v 1.500 appeared on CTAN in `fonts/psfonts/tools` in September 1995 together with Sebastian Rahtz's Unix shell scripts to automate the installation. This version included a completely revised implementation of `\latinfamily` and added the `\installrawfont` command to install ligfull raw fonts.

v 1.504 appeared on CTAN in `fonts/psfonts/tools` in January 1996. It added code to install (real or faked) small caps fonts and to fake narrow fonts, if they don't exist.

v 1.511 was one of the last beta versions by Alan Jeffrey which dates back to June 1996, but for some reason was never released. It uses the encodings `9o` and `9d` instead of `9t` and `9e` to install expertized oldstyle fonts, but this change never found its way into subsequent "unofficial" releases until v 1.8.

v 1.6 appeared on CTAN in `fonts/psfonts/tools` in February 1997 and was the first "unofficial" version of fontinst maintained by Sebastian Rahtz. It contains only minor changes compared to v 1.504, such as switching to lowercase file names, searching for `.pl` files before `.afm` files, and adding the `textcomp` installation to the distribution.

v 1.6 was re-issued on CTAN in September 1997, when the Unix shell scripts were replaced by Perl scripts. The fontinst package itself didn't change apart from minor changes of the debugging messages.

v 1.7 is the latest "unofficial version", which was released on CTAN in May 1998. It includes some changes to the mapping between `fontname` and LaTeX weight codes proposed by Ulrik Vieth, as well as some code to support the use of `8r`-encoded `.afm` files generated by `ttf2afm`.

2.2 About fontinst v 1.8

Version 1.8 of the fontinst utility was the first to have been converted to docstrip format (this was made by Ulrik Vieth) and it was first released in June 1998. It is based on Sebastian Rahtz's "unofficial" fontinst releases v 1.6 and 1.7 as of February 1997 and May 1998, which, in turn, are based on Alan Jeffrey's versions v 1.500 and 1.504 of September 1995 and January 1996. In addition, version v 1.8 also incorporated some code from Alan Jeffrey's last beta version, v 1.511, which dates back to June 1996, but was never released to the general public for some reason.

Version 1.8 was agreed upon by the fontinst hacker community, as represented on the fontinst mailing list, as the new officially endorsed version, on which all further additions and enhancements can be based.

3 About this fontinst release

Several improvements, some of which fixes old bugs and some of which introduces new features, have been made to in particular the generation of fonts. The most important are:

- The old multiple `\setkern` bug, which resulted in pointless `KRN` instructions being written to (V)PL files have been fixed. In addition, it is now possible to reset kerns (using `\resetkern`) and unset kerns (using `\unsetkerns`, `\noleftkerning`, `\norightkerning`, or `\noleftrightkerning`).
- A bug which resulted in too few kerns being written to a font if the same glyph is used in more than one slot (this occurs if you make an all-caps font from an encoding definition file which sets `\lc`, `\lctop`, and `\lclig` to the same to their `\uc`... counterparts) has been fixed. A consequence of this fix is that you cannot access the number of the slot that glyph ⟨*glyph*⟩ has been assigned to as `\int{`⟨*glyph*⟩`}` (unless you are using `cfntinst.sty`), but there really is no need for you to do so either.
- There is now a proper interface for using left and right boundaries which, in contrast to the old setting of the integer `boundarychar`, can make use of the full generality of the PL format. (The old interface is still available if you use `cfntinst.sty`.)

- fontinst can now interpret AFM files which contain non-integer dimensions. There is also a user-level interface for changing the formula according to which the italic corrections of glyphs are computed.
- The PL to MTX converter can now cope with the VPL-unique property lists `VTITLE`, `MAP`, and `MAPFONT`, which means that it can now convert VPL files as well. In addition to this, there are a couple of minor bugfixes and improvements in the PL to MTX conversion.
- There is now a way of overriding the PL to MTX converter's automatic choice of encoding definition file made based on the `CODINGSCHEME` property of a PL file. The command `\generalpltomtx` lets you select which encoding definition file you want to use, as does the new `\fromplgivenetx` transformed font command.
- Several fontinst commands that were previously missing have been added to the fontdoc package. The most significant are probably `\inputmtx` and `\inputetx`.
- There is now a simple way of changing the names of glyphs in metric files generated from AFM files.
- The implementation of font metrics transformations, as done by `\transformfont`, has been rewritten. This can result in different font metrics, but we think only in that it comes out more like one would expect. The user interface is the same.
- There is now a new transformed font command `\fromany` which searches for a font metric file in any of the four formats MTX, PL, AFM, and VPL; converting it to MTX format and possibly generating a PL as well if required. This command is also used by `\installfont`, `\installrawfont`, and `\reglyphfont` to search for font metric files, so these can now take VPL files as a source for font metric data.
- The automatic map file generation is now working! It does not yet generate a map file that any known driver can read, however, but the interface to main fontinst has been established and there is a system for opening and closing output files, writing map files for multiple drivers simultaneously, and avoiding duplicate entries. What is missing is the routines for writing the actual entries; here we hope for contributions from other fontinst users.

3.1 Metric packages

The idea of the metric package commands is to make it easier to split all the commands normally kept in `latin.mtx` on several files, which is handy if one wants to make minor modifications to some of the commands or simply leave some of the parts in `latin.mtx` out to speed up the fontmaking.[1] They are not of any use to those who use fontinst only at the `\latinfamily` level, but they should prove useful to those who are in the business of modifying `latin.mtx` or writing their own alternatives, as it simplifies modularisation.

[1] In my experience, the speedup can be considerable./LH

The main problem with splitting up `latin.mtx` (or some other metric file which fulfills an equivalent function) is that there are some commands which are defined at the top and which are then used in almost all sections of the file. One must make certain that these commands are always loaded, which makes the metric files somewhat harder to use (especially if the one who tries to use them is not the one who wrote them).

One strategy is to include all definitions needed for a metric file in it. This has the slight disadvantage that the commands will have to be definied several times. What is worse however, is that the command definitions will appear in several files, so if one finds a bug in one of them, one cannot simply correct this bug in one place. As the number of files can soon become quite large, correcting such bugs can become a boring procedure indeed.

Another strategy is to put all the command definitions in one file and then explicitly include it in the ⟨*file-list*⟩ argument of `\installfont`. This eliminates the repeated bug fixing problem, but requires the user to do something that the computer can actually do just as well.

A third strategy is to put the command definitions in one or several files and then in each metric file the user explicitly mensions load the command definitions needed for that particular file. Metric packages uses an improved version of this strategy, since they also make it possible for fontinst to remember which packages (i.e., sets of command definitions) that have already been loaded, so that they are not unnecessarily loaded again.

For information on syntax etcetera, see Section 11.

3.2 Word boundary ligatures and kerns

One of the new features added in TEX 3 was that of ligatures and kerns with word boundaries. fontinst has had an interface for making such ligatures and kerns, but it has been completely redesigned in v 1.9 and the old interface (setting the integer `boundarychar`) is no longer recognized by fontinst. Files which use the old interface can still be processed with `cfntinst.sty`, though.

Before considering the new commands, it is suitable to make a distinction between proper glyphs and pseudoglyphs. A proper glyph has been set using one of the commands `\setrawglyph`, `\setglyph`, and `\resetglyph`. A pseudoglyph is any name used in the context of a glyph name which does not denote a proper glyph. If a pseudoglyph `g-not` was set using the `\setnotglyph` command, the if `\ifisglyph{g-not}\then` will evaluate to true, but something can be a pseudoglyph even if an `\ifisglyph` test evaluates to false. The interesting point about pseudoglyphs when considering word boundaries however, is that a pseudoglyph can have ligatures and kerns.

Kerns and ligatures at the left word boundary (beginning of word) are specified using the commands `\setleftboundary` and `\endsetleftboundary`, which are syntactically identical to `\setslot` and `\endsetslot` respectively. One important difference is however that the argument to `\setslot` must be a proper glyph, while the argument to `\setleftboundary` may be any glyph, hence any pseudoglyph will do just fine.

`\ligature` commands between `\setleftboundary` and `\endsetleftboundary` will generate beginning of word ligatures. Kerns on the right of the glyph specified in `\setleftboundary` will become beginning of word kerns.

Kerns and ligatures at the right word boundary (end of word) are trickier, due to the asymmetrical nature of the ligkern table in a PL file. What a font can do is to specify that the right word boundary, for purposes of kerning and ligatures, should be interpreted as character n. By including a kern or ligature with character n on the right, that kern or ligature will be used at the end of a word, but it will also be used each time the next character is character n. Because of this, one usually wants the slot n, which the right word boundary is interpreted as being, to be empty whenever the encoding allows this.

The command

`\setrightboundary{`⟨*glyph*⟩`}`

will mark the current slot as used to denote the right word boundary, and leave the slot empty, increasing the current slot number by one just like a `\setslot` ... `\endsetslot` block does. Kerns on the left of ⟨*glyph*⟩ will be end of word kerns and `\ligature` commands with ⟨*glyph*⟩ as the second argument will be for the end of a word.

The command

`\makerightboundary{`⟨*glyph*⟩`}`

is similar to `\setrightboundary`, but it is a slot command which may only be used between a `\setslot` and the matching `\endsetslot`. Like `\setrightboundary`, it marks the current slot as used to denote the right word boundary, but the glyph specified in the enclosing `\setslot` will be written to that slot. Ligatures for the glyph specified by the `\setslot` and ligatures for the glyph specified by the `\makerightboundary` will both be for this single slot. Kerns on the right of the `\setslot` glyph and the `\makerightboundary` glyph will similarly both be for this single slot. The idea is that the `\setslot` glyph should be used when making a kern or ligature for that glyph, while the `\makerightboundary` glyph should be used when making a kern or ligature for the end of a word. fontinst will warn you if these two uses of the slot directly contradict each other.

3.3 Changing the names of glyphs

Sometimes, primarily when making a virtual font from more than one raw font and two of the raw fonts contain different glyps with the same name, it becomes necessary to change the names of some glyphs to make some sense out of it. The main source of this kind of trouble is the "caps and small caps" (CSC) and "oldstyle figures" (OsF) fonts within many commercial font families. The typical problem is that what is typographically different glyphs—such as the lowercase 'a' (`a`, for fontinst) and the smallcaps 'A' (`Asmall`, for fontinst)—are given the same name by the foundry.

One way to get round this is to say for example

```
\setglyph{Asmall} \glyph{a}{1000} \endsetglyph
\setleftrightkerning{Asmall}{a}{1000}
\unsetglyph{a}
\noleftrightkerning{a}
```

and continuing like that for all the duplicate glyph names. This is however a rather prolix method and if the number of glyphs is large then it is usually simpler to use the `\reglyphfont` command.

To reglyph one or several fonts, one writes

```
\reglyphfonts
  ⟨reglyphing commands⟩
\endreglyphfonts
```

There are two types of reglyphing commands: the `\reglyphfont` command, and the commands that modify what `\reglyphfont` will do to the fonts it operates on. The syntax of `\reglyphfont` is

```
\reglyphfont{⟨destination font⟩}{⟨source font⟩}
```

The ⟨*source font*⟩ font here is the name (suffix not included, of course) of the font metric file one wants to change the glyph names in. This font metric file can be in any of the formats MTX, PL, AFM, and VPL, and it will be converted to MTX format if it isn't already in that format (this happens just as for files listed in the second argument of `\installfont`). ⟨*destination font*⟩ (which must be different from ⟨*source font*⟩) will be taken as the name for a new `.mtx` file that will be generated. The destination font can differ from the source font only in two ways: the names of some glyphs in the source font might be changed, and some of the commands from the source font might not have been copied to the destination font. To what extent the fonts are different is determined by what modifying commands have been executed; when no modifying commands have been executed, the source and destination font are equal.

The modifying reglyphing commands are

```
\renameglyph{⟨to⟩}{⟨from⟩}
\renameglyphweighted{⟨to⟩}{⟨from⟩}{⟨weight⟩}
\killglyph{⟨glyph⟩}
\killglyphweighted{⟨glyph⟩}{⟨weight⟩}
\offmtxcommand{⟨command⟩}
\onmtxcommand{⟨command⟩}
```

`\renameglyph` simply declares that occurencies of the glyph name ⟨*from*⟩ should be replaced by the glyph name ⟨*to*⟩. To each glyph name is also assigned a *weight*, which is used by a mechanism which conditions copying of commands from the source font to the destination font by the set of glyphs that command mensions. The details of this mechanism are however somewhat tricky, so those interested in the full generality should go to Subsection 16.3. Here it needs only be noted that if one applies `\killglyph` to a glyph name, then (under most circumstances) commands that refer to that glyph name will not be copied to the destination font.

`\offmtxcommand` and `\onmtxcommand` also control whether commands are copied to the destination font, but they look at the actual command rather than the glyphs it refers to. For example, after the command

```
\offmtxcommand{\setkern}
```

no `\setkern` commands will be copied. By using `\offmtxcommand`, it is possible to achieve effects similar to those of the files `kernoff.mtx` and `glyphoff.mtx`—the difference is that with `\offmtxcommand`, it happens at an earlier stage of the font generation. As expected, `\onmtxcommand` undoes the effect of `\offmtxcommand`.

The effects of modifying reglyphing commands are delimited by `\reglyphfonts` and `\endreglyphfonts`, which starts and ends a group respectively.

As we expect the most common reglyphing operation will be to go from CSC glyph names to expert glyph names, there is a file `csc2x.tex` in the fontinst distribution which contains the modifying reglyphing commands needed for setting up that conversion. Thus you can write for example

```
\reglyphfonts
  \input csc2x
  \reglyphfont{padrcx8r}{padrc8r}
  \reglyphfont{padscx8r}{padsc8r}
\endreglyphfonts
```

to alter the glyph names in the CSC fonts in the Adobe Garamond (`pad`) family. Note that the names of the destination fonts here really are rather arbitrary, since they will only exist as `.mtx` files, and thus only need to work within your local file system. In particular, all the `\setrawglyph` commands in the destination font files still refer to the source font, so it is that font which the drivers need to know about.

3.4 Making map file fragments

The mechanism that lets fontinst write the fragments of a DVI driver map file the driver would need to handle a set of fonts generated by fontinst is still under construction, but it has now reached the level where it can be helpful. The goal is that the map file fragment writer should be able to write the lines[2] of a map file that the driver would need for handling the raw fonts fontinst has (i) installed directly (using `\installrawfont`) or (ii) used as a base font for some installed virtual font (generated by `\installfont`). There is still some work to do before that goal is met, however.

What the map file fragment writer *can* do today is that it can tell you

- which fonts there need to be entries in the map file for,
- for each font what metric transformations (scaling, slanting, etc.) need to be applied,
- for each font whether it must be reencoded and in that case to what encoding,
- for most fonts what the postscript name for this font is (you'll probably need this if the driver is to generate postscript or PDF output).

You still have to edit it all into some format that your driver will understand, but at least you will be spared the work of collecting the information.

Now what does one have to do to use this map file fragment writer, then? First you need to tell fontinst to record the information the map file fragment writer needs. You do this by giving the command

```
\recordtransforms{whatever.tex}
```

at the beginning of the run. Here `whatever.tex` is the name of a file that will be created, so you can use some other name if you like. After that you do all the calls to `\transformfont`, `\installfont`, `\installrawfont`, `\latinfamily`, etc. you need to make the fonts you want. When you're done, you give the command

[2] Not in general an entire map file, hence the word *fragment*.

```
\endrecordtransforms
```

and end the run (say `\bye`). The file `whatever.tex` will now contain the information about which fonts were used and what needs to be done with them.

The second step is to actually run the map file fragment writer. Observe that it is located in the file `finstmsc.sty`, not `fontinst.sty`! The commands you need to give it can be so few that you can type them in at TEX's `*` prompt, but if you are writing a command file then it should typically have the following structure (comments not necessary, of course):

```
\input finstmsc.sty                    % Input command definitions
⟨general settings⟩                     % See below
\adddriver{debug}{whatever.map}        % Open output file
\input whatever.tex                    % Write to output file
\donedrivers                           % Close output file(s), tidy up
\bye                                   % Quit
```

The `\adddriver` command gives the order "write map file entries for the ⟨*first argument*⟩ DVI driver to the file ⟨*second argument*⟩." The plan is that it should be possible to use the name of just about any major driver (`dvips`, `xdvi`,[3] `pdftex`, `OzTeX`, etc.) here and get suitable map file entries for that driver as output, but for the moment the only acceptable value for the first argument is `debug`. The entries in the file that is written for the `debug` "DVI driver" simply contain all the available information about each font (hence it should come handy for debugging code writing entries for real drivers) in a format that should be easy to interpret for a human.

The file `whatever.tex` in the above example contains the commands (`\makemapentry` commands) that actually cause entries to be written to the output file. It also contains a number of `\storemapdata` commands—these describe how some given font was made. One problem with the current setup is that if the MTX file for some font was made before the `\recordtransforms` command was given, perhaps on an earlier fontinst run, then the file `whatever.tex` above will not contain a `\storemapdata` for that font, and in this case subsequent `\makemapentry`s won't know what to make of it, so you'll get an error message. There are two ways to avoid this: (i) include a `\storemapdata` for each font made before `\recordtransforms` in the command file, for example in the ⟨*general settings*⟩ part, or (ii) see to that you only use fonts that have their MTX files generated after the `\recordtransforms` command. fontinst makes it a little tricky to use (ii), since MTX files usually won't be generated if one with the right name already exists. To be on the safe side here, you'll have to delete all MTX files generated from AFM, PL, VPL, or other MTX files between each fontinst run. We plan to have that annoying detail fixed by the time the next fontinst release is made, however.

Another class of things that will typically appear in the ⟨*general settings*⟩ part above is commands that will inform the routines actually writing output about characteristics of your TEX system, of the set of fonts you are using on this run, or of something else that might be useful. For the moment, there are three commands of this type, all of which has to do with what should be used for the postscript font name of a font whose metrics are taken from a PL file.

`\AssumeMetafont` Assume all fonts with PL metrics are bitmaps generated by Metafont, and therefore make no entries for them.

[3] Or does that use the same map file as `dvips`? I heard somewhere that it did. /LH

`\AssumeAMSBSYY` Assume all fonts with PL metrics have their TEX names in all upper case as postscript names—just like the Computer Modern fonts in the AMS/Blue Sky/Y&Y distribution.

`\AssumeBaKoMa` Assume all fonts with PL metrics have their TEX names in all lower case as postscript names—just like the Computer Modern fonts in the BaKoMa distribution.

Otherwise the default action is that the routine for finding out this simply observes that it hasn't got a clue about what the right value is when the metrics were taken from a PL file, and therefore this field comes out as '`??????`' in those cases.

3.5 On monowidth and typewriter fonts

Traditionally, the `monowidth` integer has controlled the aspects of the behaviour of fontinst's standard metric and encoding files that have to do with monowidth and typewriter fonts. Unlike all the other integers, the value of this integer did not matter, only whether the integer was set or not. The user typically set this integer outside an `\installfonts` ... `\endinstallfonts` block (if it was to be set during all font installations in that block), or in the first file in the file-list argument of `\installfont` or `\installrawfont` (if it should only be set for a few files installed in that block).

The effects of setting this integer has been rather diverse and the only way to find out exactly which they are for any given setup is to read the metric and encoding files in question. The general idea is however that everything in a monowidth font should have width a multiple of the width of an ordinary letter, so that characters on different rows will always align horizontally. This means for example that spaces of such fonts will get zero stretch and shrink, since that would otherwise completely mess up the alignment.

However, some of the things `monowidth` has used to control really was not about the font being a monowidth font, but rather about the font being a typewriter font. There is a large covariance between these two concepts, but they are not equivalent, and hence code that is conditioned by these two attributes should not be controlled through a single switch. Therefore a second variable `typewriter` has been introduced to control the typewriter aspects of what metric and encoding files do.

To get the old effect of setting `monowidth`, one should now set both `monowidth` and `typewriter`. To give some support to old code, there is also code which will set `typewriter` if it isn't set and `monowidth` is set to anything but 9876. The code that does this will however disappear eventually, and it always prints a warning message on the terminal.

The above may give the impression that the change to `typewriter` taking over some of the functions of `monowidth` has completely taken place, but this is not the case. To date, only the files `ot1.etx`, `8r.etx`, and `8y.etx` have been updated this way, and even though all encoding and metrics files that are affected by typewriter/monowidth characteristics should eventually be updated, the change will probably be slow and for a few files at a time. The process of introducing this distinction also poses some questions which seem suited for a public debate. One of these which are especially noteworthy is which effects `typewriter` should have in `t1.etx`, since this file has previously ignored typewriter aspects of fonts.

4 Notes on using fontinst

The primary purpose of fontinst is to simplify the installation of PostScript or TrueType text fonts.

4.1 General notes

Leaving aside unusual variants which require special attention such as alternate or swash fonts, almost all standard font families can be installed automatically using the `\latinfamily` command, optionally making use of the corresponding expert fonts as well.

Depending on what kind of fonts you have and want to install, `\latinfamily` supports three different modes of operation:

`\latinfamily{<fam>}{}` installs a normal font family using `8a`-encoded standard fonts (reencoded to `8r`) and nothing else. It installs `.fd` files for the LaTeX families `8r<fam>`, `OT1<fam>`, `T1<fam>` and `TS1<fam>`, and generates virtual fonts for the `7t`, `8t` and `8c` encodings. This is the only option available for most typefaces which do not have an expert set.

`\latinfamily{<fam>x}{}` installs an expertized font family using `8a`-encoded standard fonts (reencoded to `8r`) and `8x`-encoded expert fonts. It installs `.fd` files for the LaTeX families, `OT1<fam>x`, `T1<fam>x` and `TS1<fam>x`, and generates virtual fonts for the `9t`, `9e` and `9c` encodings.

`\latinfamily{<fam>j}{}` installs an expertized font family with oldstyle digits using `8a`-encoded standard fonts (reencoded to `8r`) and `8x`-encoded expert fonts. It installs `.fd` files for the LaTeX families, `OT1<fam>j`, `T1<fam>j` and `TS1<fam>j`, and generates virtual fonts for the `9o`, `9d` and `9c` encodings. Since TS1 has oldstyle digits by default, the `9c`-encoded fonts should be the same as in the previous case. Finally, `\latinfamily{<fam>9}{}` is also supported as an alternative to `\latinfamily{<fam>j}{}` for backwards compatibility.

The whole installation process relies on certain assumptions about the symbol complement of Adobe's expert fonts. In particular, it is assumed that the expert fonts include the oldstyle digits and a complete set of small caps glyphs, which is an assumption that's not always statisfied for expert fonts by other suppliers. If these glyphs are not included in the expert fonts, the only way to get them is from real small caps fonts, but this requires some reshuffling of glyph names.

To support such unusual cases, this source file contains some optional code embedded between `<*oldstyle>` ... `</oldstyle>`, which extends the behaviour of `\latinfamily` for expertized encodings with oldstyle digits. Instead of relying only on the glyphs of the `8r`-reencoded raw font and the `8x`-encoded expert font, this version also looks for corresponding OsF or SC&OsF fonts and uses the default digits from those fonts as oldstyle digits.

4.2 Notes on small caps and olstyle fonts

The `\latinfamily` command is supposed to do a reasonably good job of installing a complete font family based on all the `.afm` files it can find. If it doesn't find a suitable font shape, it is sometimes possible to fake it by default substitutions. However, in the case of small caps fonts, there are several options which may require some clarification.

For the majority of typefaces, a font family typically consists only of of roman and italic fonts in several weights. Since real small caps fonts are not included, they have to be faked from the roman fonts, which is implemented by setting `\encoding_shape` to '`c`', so that different encoding files `OT1c.etx` or `T1c.etx` are used. Since these files call for glyph names such as '`Asmall`' which are not found in the roman font, the default substitutions in `latin.mtx` are eventaully used to approximate fake small caps glyphs by scaling and letterspacing. The outcome is just an approximation for a small caps font, but it is better than nothing.

For a small number of typefaces, the standard fonts are complemented by an expert collection, which ususally includes two sets of fonts. First, for each standard font there is a corresponding expert font containing additional glyphs such as extra ligatures and symbols, oldstyle digits and small capital letters. For Adobe expert fonts this set is sufficient to build a complete small caps font from the standard and expert glyphs. Furthermore, the expert collection usually also contains a number of real small caps font corresponding to the roman fonts and some OsF fonts corresponding to the italic fonts. If these fonts are available, there are several options how to install small caps fonts.

By default, `\latinfamily` first tries to find a real small caps font. If it is found, it is installed using the default encoding files and metric files, just like any roman or slanted font. However, once `\latinfamily` has taken this choice, it will fail to find a corresponding expert font, since it is actually looking for an expert font in small caps shape which doesn't exist. (In fact, it would be an error to substitute an expert font in normal shape.) The outcome will be a virtual font based only on glyphs from the real small caps raw font, which implies ending up with oldstyle digits as the default set of digits, but allows to inherit the kerning information of the real small caps font.

Another option for an expertized installation would be to make the real small caps fonts unavailable, so that `\latinfamily` will attempt to fake a small caps font using glyphs from the standard and expert fonts in normal shape. This means that `\encoding_shape` is again set to '`c`', so that `OT1c.etx` and `T1c.etx` are used, but this time a glyph named '`Asmall`' does exist in the expert font and will be used instead of faked one generated by scaling. The outcome will be a font based on normal and small caps glyphs from the standard and expert fonts. The oldstyle digits will only be used if they are called for, otherwise the default digits from the roman font are used. The only drawback of this approach is that the kerning around small capital letters will be based on the scaled kern amounts of the capital letters rather than on the kern pairs from the real small caps font.

Finally, the most promising approach of all these options would be to combine the glyphs from standard and expert fonts with kern pairs from the real small caps fonts. The current version of `\latinfamily` does not implement this, but it would be a worthwile approach, and advanced fontinst users are encouraged to attempt it. The file `csckrn2x.tex` (a variant of `csc2x.tex`, which is mensioned in Subsection 3.3) should come in handy for this.

5 fontdoc package overview

The purpose of the fontdoc package is to support typesetting of `.etx` and `.mtx` files intended for use with fontinst. The typical format of these files looks something like this:

```
\relax
\documentclass[twocolumn]{article}
\usepackage{fontdoc}

\begin{document}
LaTeX commands
\encoding or \metrics
fontinst commands
\endencoding or \endmetrics
LaTeX commands
\end{document}
```

To make it work, fontdoc has to define all the user-level fontinst commands in terms of typesetting instructions. This goal is currently only partially achieved, but the percentage of fontinst commands covered by fontdoc is still fairly large, so it is our impression that problems with commands not covered occur only very rarely.

File a
fibasics.dtx

6 The first and the last

6.1 Version numbers

We start by making some default catcode assignments, in case we are using iniTEX.

```
⟨*pkg⟩
\catcode`\{=1
\catcode`\}=2
\catcode`\#=6
\catcode`\^=7
⟨/pkg⟩
```

`\fontinstversion` If we are running under iniTEX we cannot put the identification stuff any earlier than this. Note that `\fontinstversion` is not just used for indentification but also in `\needsfontinstversion`.

```
⟨*pkg | doc⟩
\def\fontinstversion{1.910}
\def\filedate{1999/11/04}
⟨/pkg | doc⟩
```

If we are running under iniTEX or `plain`, we have to get around the LaTeX-specific `\ProvidesPackage` stuff.

```
⟨*pkg | doc⟩
\ifx\ProvidesPackage\undefined
   \def\NeedsTeXFormat#1{}
   \def\ProvidesPackage#1[#2]{}
\fi
⟨/pkg | doc⟩
```

Now we can identify ourselves as usual.

```
⟨doc⟩\ProvidesPackage{fontdoc}
⟨pkg⟩\ProvidesPackage{fontinst}
⟨pkg | doc⟩[\filedate\space v\fontinstversion\space
⟨doc⟩  fontinst documentation package]
⟨pkg⟩  fontinst installation package]
```

> "Can't say I've ever been too fond of beginnings, myself. *Messy* little things. Give me a good ending any time. You know where you *are* with an ending."
>
> — from *The kindly ones* by Neil Gaiman et al.

While we're at version numbers anyway, we might as well define the command for testing them. Note however that the module name is not ⟨pkg⟩ as above, but ⟨pkg2⟩. Code in the latter module ends up at the very end of `fontinst.sty`!

`\needsfontinstversion` The macro:

`\needsfontinstversion{`⟨*number*⟩`}`

checks the version number.

```
⟨*pkg2⟩
\def\needsfontinstversion#1{{
   \a_dimen=#1pt
   \b_dimen=\fontinstversion~pt\x_relax
   \ifnum\a_dimen>\b_dimen
      \immediate\write16{}
      \immediate\write16{Warning:~This~file~needs~fontinst~version~#1.}
      \immediate\write16{Warning:~You~are~using~version~
         \fontinstversion.}
      \immediate\write16{Warning:~This~may~cause~errors.}
      \immediate\write16{}
   \fi
}}
⟨/pkg2⟩
```

In fontdoc, `\needsfontinstversion` is printed out as a comment.

```
⟨*doc⟩
\def\needsfontinstversion#1{\Bheading{Needs fontinst v\thinspace#1}}
⟨/doc⟩
```

6.2 Special catcodes and configuration file

fontinst uses some unusual, but convenient, settings of `\catcode`. `@` and `_` are made letters, `~` is made a space, and space and newline are ignored. Before setting those however, we save the current values of the catcodes, so that they can be restored at the end of `fontinst.sty`.

```
⟨*pkg⟩
\edef\spacecatcode{\the\catcode`\ }
\edef\nlcatcode{\the\catcode`\^^M}
\edef\atcatcode{\the\catcode`\@}
\edef\underscorecatcode{\the\catcode`\_}
\edef\tildecatcode{\the\catcode`\~}
\catcode`\ =9
\catcode`\^^M=9
\catcode`\@=11
\catcode`\_=11
\catcode`\~=10
⟨/pkg⟩
```

We input the `fontinst.rc` file, if it exists. (Search order changed 1997/02/07 by Thierry Bouche.)

UV, 06/1998: What is this `fontinst.rc` file good for? It turns out that you can use it to do `\def\raw_encoding{8y}` if you prefer to install your fonts the other way.

```
⟨*pkg2⟩
⟨*!misc⟩
\if_file_exists{fontinst.rc}\then
   \primitiveinput fontinst.rc
\else
   \immediate\write16{No~file~fontinst.rc.}
\fi
⟨/!misc⟩
```

Use a different configuration file for `finstmsc.sty`.

```
⟨*misc⟩
\if_file_exists{finstmsc.rc}\then
   \primitiveinput finstmsc.rc
\else
   \immediate\write16{No~file~finstmsc.rc.}
\fi
⟨/misc⟩
```

At the end of `fontinst.sty`, we restore the catcodes we changed.

```
\catcode`\@=\atcatcode
\catcode`\^^M=\nlcatcode
\catcode`\ =\spacecatcode
\catcode`\~=\tildecatcode
\catcode`\_=\underscorecatcode
⟨/pkg2⟩
```

6.3 Plain TEX macros from `fontinst.ini`

If we're running in iniTEX we input some definitions taken from `plain`.

```
⟨*pkg⟩
\ifx\@ne\undefined_command
   \input fontinst.ini\relax
\fi
⟨/pkg⟩
```

```
⟨*ini⟩
\chardef\active=13

\chardef\@ne=1
\chardef\tw@=2
\chardef\thr@@=3
\chardef\sixt@@n=16
\chardef\@cclv=255
\mathchardef\@cclvi=256
\mathchardef\@m=1000
\mathchardef\@M=10000
\mathchardef\@MM=20000

\count10=22 % allocates \count registers 23, 24, ...
\count11=9 % allocates \dimen registers 10, 11, ...
\count15=9 % allocates \toks registers 10, 11, ...
\count16=-1 % allocates input streams 0, 1, ...
\count17=-1 % allocates output streams 0, 1, ...
\count20=255 % allocates insertions 254, 253, ...
\countdef\insc@unt=20 % the insertion counter
\countdef\allocationnumber=21 % the most recent allocation
\countdef\m@ne=22 \m@ne=-1 % a handy constant
\def\wlog{\immediate\write\m@ne} % write on log file (only)

\countdef\count@=255
\dimendef\dimen@=0

\outer\def\newcount{\alloc@0\count\countdef\insc@unt}
```

```
\outer\def\newdimen{\alloc@1\dimen\dimendef\insc@unt}
\outer\def\newtoks{\alloc@5\toks\toksdef\@cclvi}
\outer\def\newread{\alloc@6\read\chardef\sixt@@n}
\outer\def\newwrite{\alloc@7\write\chardef\sixt@@n}

\def\alloc@#1#2#3#4#5{\global\advance\count1#1by\@ne
  \ch@ck#1#4#2% make sure there's still room
  \allocationnumber=\count1#1%
  \global#3#5=\allocationnumber
  \wlog{\string#5=\string#2\the\allocationnumber}}
\outer\def\newinsert#1{\global\advance\insc@unt by\m@ne
  \ch@ck0\insc@unt\count
  \ch@ck1\insc@unt\dimen
  \ch@ck2\insc@unt\skip
  \ch@ck4\insc@unt\box
  \allocationnumber=\insc@unt
  \global\chardef#1=\allocationnumber
  \wlog{\string#1=\string\insert\the\allocationnumber}}
\def\ch@ck#1#2#3{\ifnum\count1#1<#2%
  \else\errmessage{No~room~for~a~new~#3}\fi}

\outer\def\newif#1{\count@\escapechar \escapechar\m@ne
  \expandafter\expandafter\expandafter
   \edef\@if#1{true}{\let\noexpand#1=\noexpand\iftrue}%
  \expandafter\expandafter\expandafter
   \edef\@if#1{false}{\let\noexpand#1=\noexpand\iffalse}%
  \@if#1{false}\escapechar\count@} % the condition starts out false
\def\@if#1#2{\csname\expandafter\if@\string#1#2\endcsname}
{\uccode`1=`i \uccode`2=`f \uppercase{\gdef\if@12{}}} % `if' is required

\newdimen\p@ \p@=1pt % this saves macro space and time
\newdimen\z@ \z@=0pt % can be used both for 0pt and 0

\def\space{ }
\let\bgroup={
\let\egroup=}

\def\loop#1\repeat{\def\body{#1}\iterate}
\def\iterate{\body \let\next\iterate \else\let\next\relax\fi \next}
\let\repeat=\fi % this makes \loop...\if...\repeat skippable

\def\superject{\par\penalty-\@MM}
\outer\def\bye{\par\vfill\superject\end}
```
⟨/ini⟩

The two last commands above also need to be defined if we're running under LaTeX. In that case we need to be careful with `\bye` however, as no `\if...` can skip over code that explicitly contains that control sequence, if it has been defined as above.

```
⟨*pkg⟩
\ifx\bye\undefined_command
   \def\superject{\par\penalty-\@MM}
   \outer\expandafter\def \csname bye\endcsname
     {\par\vfill\superject\@@end}
```

```
\fi
⟨/pkg⟩
```

```
⟨*ini & everyjob⟩
\everyjob{%
   \if_file_exists{fontinst.rc}\then
      \primitiveinput~fontinst.rc
   \else
      \immediate\write16{No~file~fontinst.rc.}
   \fi
}
⟨/ini & everyjob⟩
```

7 Basic definitions

7.1 Declaring variables and constants

Some temporary variables:

```
⟨*pkg⟩
\newcount\a_count
\newcount\b_count
\newcount\c_count
\newcount\d_count
\newcount\e_count
\newcount\f_count
\newcount\g_count
```

```
\newdimen\a_dimen
\newdimen\b_dimen
\newdimen\c_dimen
\newdimen\d_dimen
```

```
\newtoks\a_toks
\newtoks\b_toks
\newtoks\c_toks
```

```
\newif\if_a_
```

Besides these temporary variables that have to be declared, the family of control sequences with names of the form \⟨*letter*⟩`_macro`, such as `\a_macro`, `\b_macro`, etc., should be used as "macro valued" temporary variables.

`\out_file` Some global variables:

```
\newwrite\out_file
```

Some constants:

```
\mathchardef\one_thousand=1000
\mathchardef\five_hundred=500
\mathchardef\one_hundred=100
\mathchardef\max_mathchardef="7FFF
```

7.2 Trigonometry macros

Include David Carlisle's trigonometry macros. First, do some hacks to get round all that `\NeedsTeXFormat{LaTeX2e}` stuff that David put in :-) In v 1.8 these hacks been moved to an earlier place since we put in `\ProvidesPackage` ourselves.

```
\let\@tempdima\a_dimen
\let\@tempdimb\b_dimen
\input trig.sty
⟨/pkg⟩
```

7.3 fontdoc-specific declarations

Two global variables:

```
⟨*doc⟩
\newdimen\a@dimen
\newcount\a@count
```

Some useful macros and constrol structures:

```
\def\x@cs#1#2{\expandafter#1\csname#2\endcsname}
```

`\Aheading` `\Bheading` `\Aheading` and `\Bheading` are used to typeset various headings.

```
\def\Aheading#1{%
    \par\addvspace\medskipamount
    \noindent\textbf{#1}\vadjust{\penalty200}\par
}
\def\Bheading#1{\par\noindent{\bf#1}}
```

Some formating commands and a macro that comes in handy:

```
\raggedright
\raggedbottom
\def\plain{{\tt plain}}
```

`\plainint` `\plaindiv` fontdoc saves away Plain TeX's `\int` and `\div` as `\plainint` and `\plaindiv`, since we are going to use the control sequences for other purposes.

```
\let\plainint=\int
\let\plaindiv=\div
⟨/doc⟩
```

8 TeX hackery

8.1 Utiltiy macros

`\x_cs` `\x_relax` `\g_let`

```
⟨*pkg⟩
\def\x_cs#1#2{\expandafter#1\csname#2\endcsname}
\let\x_relax=\relax
\def\g_let{\global\let}
⟨/pkg⟩
```

`\x@relax`

```
⟨doc⟩\let\x@relax=\relax
```

\empty_command
\gobble_one
\gobble_two
\gobble_three
\identity_one
\first_of_two
\second_of_two

```
⟨*pkg⟩
\def\empty_command{}
\def\gobble_one#1{}
\def\gobble_two#1#2{}
\def\gobble_three#1#2#3{}
\def\identity_one#1{#1}
\def\first_of_two#1#2{#1}
\def\second_of_two#1#2{#2}
```

\hash_char
\percent_char
\left_brace_char
\right_brace_char

```
\bgroup
   \catcode`\[=1
   \catcode`\]=2
   \catcode`\#=12
   \catcode`\%=12
   \catcode`\{=12
   \catcode`\}=12
   \gdef\hash_char[#]
   \gdef\percent_char[%]
   \gdef\left_brace_char[{]
   \gdef\right_brace_char[}]
\egroup
```

\lose_measure Get rid of a trailing `pt` when converting dimension.

```
{
   \catcode`\p=12
   \catcode`\t=12
   \gdef\lose_measure#1pt{#1}
}
```

\first_char Return the first character of a string.

```
\def\first_char#1#2={#1}
```

\add_to Append one or more tokens to the replacement text of a parameterless macro.

```
\def\add_to#1#2{
   \ifx#1\x_relax
      \def#1{#2}
   \else
      \expandafter\def\expandafter#1\expandafter{#1#2}
   \fi
}
```

\prep_to Prepend one or more tokens to the replacement text of a parameterless macro. Note that if more than one token is added then the second parameter must contain an `\expandafter` between every pair of tokens you actually mean to contribute. Thus if you want to prepend `abc` to `\next`, you must write

```
\prep_to\next{a\expandafter b\expandafter c}
```

Also note that the second argument must not be empty.

```
\def\prep_to#1#2{
   \ifx#1\x_relax
```

```
      \expandafter\def\expandafter#1\expandafter{\expandafter#2}
   \else
      \expandafter\def\expandafter#1\expandafter{\expandafter#2#1}
   \fi
}
```

\never_do The command \do, protected from expansion.

```
\def\never_do{\noexpand\do}
```

8.2 Writing to output files

LH 1999/03/04: As of v 1.901, there are two different output file models in fontinst. One has been along "forever" and is for writing output to files which are only open for a short period of time—every fontinst file command that opens a file in this model must also close it—and it can only be used for one file at at time. The other model offers pooled allocation of output files—as long as there is an unused TEX output stream to open, you may open a new file, and it does not matter if files are not closed in the revese order of that in which they were opened.

\open_out
\close_out
\out_line
\out_lline
\out_llline

These macros implement the classical output file commands.

```
\def\open_out#1{
   \immediate\openout\out_file=#1 \xdef\out_filename{#1}}
\def\close_out#1{
   \immediate\write16{#1~written~on~\out_filename.}
   \immediate\closeout\out_file}
\def\out_line#1{\immediate\write\out_file{#1}}
\edef\out_lline#1{\noexpand\out_line{\space\space\space#1}}
\edef\out_llline#1{
   \noexpand\out_line{\space\space\space\space\space\space#1}
}
```

\pout_line

In the new model, the basic writing command \pout_line takes two arguments: a file identifier control sequence and the token sequence to write. Thus its call looks like

\pout_line⟨*identifier*⟩{⟨*text*⟩}

The ⟨*identifier*⟩ is usually a chardef token whose number tells which input stream is used, but if TEX runs out of output streams then it might be defined as a parameterless macro which expands to \m@ne. This has the effect of making all output to that "file" go to the log file, from which the data might be salvaged.

\out_filename-STREAM

The new model operates using the control sequence family \out_filename-⟨*stream*⟩ both for storing the name of the output file and for handling allocation of output streams.

\TeX_terminal
\closed_stream
\out_filename-99
\out_filename--1

Output streams 16 and above always write to TEX's terminal, and any output file that gets closed gets its identifier set to 99 so that any attempts to write to a closed file can be easily spotted. (99 is the character code for c.) Output stream −1 is the log file, and if another output stream cannot be allocated then attempts to write to the output file will be redirected to the log file.

```
\def\TeX_terminal{\string\TeX\space terminal}
\chardef\closed_stream=99
\x_cs\let{out_filename-99}=\TeX_terminal
\x_cs\def{out_filename--1}{\jobname.log}
```

\allocate_stream
\ch@ck
```
\def\allocate_stream{
   \a_count=\m@ne
   \b_count=\m@ne
   \loop \ifnum \count17>\a_count
```
While `\a_count` < the number of the last allocated output stream ...
```
      \advance \a_count \@ne
      \x_cs\ifx{out_filename-\the\a_count}\TeX_terminal
```
If `\out_filename-`⟨*stream*⟩, where ⟨*stream*⟩ is the value of `\a_count`, is equal to `\TeX_terminal`, then output stream `\a_count` is allocated to the pool but is not used for any currently open file. Thus we've found a stream that can be used.
```
         \b_count=\a_count
         \a_count=\count17
      \fi
   \repeat
   \ifnum \b_count=\m@ne
```
In this case all streams allocated to the pool are currently in use, so try to allocate a new one.
```
      \global\advance\count17by\@ne
      \ifnum \count17<\sixt@@n
```
Then case: There was another output stream.
```
         \b_count=\count17
         \global\x_cs\let{out_filename-\the\b_count}\TeX_terminal
         \wlog{\string\write\the\b_count\space allocated~to~the~pool.}
      \else
```
Else case: See to that the output stream allocation register holds at 16.
```
         \global\count17=\sixt@@n
      \fi
   \fi
}
```
By now, if a new stream could be allocated then the number of that stream is in `\b_count`, and if it couldn't then `\b_count` is –1.

All output streams up to and including that whose number is in `\count17` is checked to see if it is available. Many unsuccessful `\newwrite`s could therefore mean we have to do quite a lot of checking. What's more, output stream 99 would incorrectly be interpreted as being in the pool but not used. To guard against these (improbable) errors, `\ch@ck` is redefined to stop allocation `\count` registers at their limit value.
```
\def\ch@ck#1#2#3{
   \ifnum \count1#1<#2 \else
      \errmessage{No~room~for~a~new~#3}
      \global\count1#1=#2
   \fi
}
```
\open_pout
\close_pout
The syntax for `\open_pout` is

`\open_pout`⟨*identifier*⟩`{`⟨*name*⟩`}`

⟨*name*⟩ is the name of the output file one wishes to open. ⟨*identifier*⟩ is a control sequence which will be redefined (globally) to act as an identifier of the file.

```
\def\open_pout#1#2{
   \allocate_stream
   \ifnum \b_count=\m@ne
      \immediate\write\sixt@@n{fontinst~is~out~of~output~streams.^^J
         Output~file~#2~cannot~be~opened.^^J
         Writes~will~be~redirected~to~the~log~file.
      }
      \gdef#1{\m@ne}
   \else
      \immediate\openout\b_count=#2\x_relax
      \x_cs\xdef{out_filename-\the\b_count}{#2}
      \global\chardef#1=\b_count
   \fi
}
```

The syntax for `\close_pout` is

`\close_pout`⟨*identifier*⟩`{`⟨*what*⟩`}`

⟨*identifier*⟩ is the output stream identifier which should have been defined in an earlier call of `\open_pout`. ⟨*what*⟩ is a string which describes what has been written to the output file, e.g. `Metrics` or `Raw~font`. It will be used to write a message on the terminal.

```
\def\close_pout#1#2{
   \ifnum #1=\closed_stream
      \errmessage{Output~file~\string#1~(#2)~not~closed,^^J
         since~it~was~not~open}
   \else
      \immediate\write\sixt@@n{#2~written~on~
         \csname out_filename-\the#1\endcsname.}
      \ifnum #1=\m@ne \else
         \immediate\closeout#1
         \global\x_cs\let{out_filename-\the#1}=\TeX_terminal
      \fi
      \global\chardef#1=\closed_stream
   \fi
}
```

`\pout_line` The syntax for `\pout_line` is

`\pout_line`⟨*identifier*⟩`{`⟨*text*⟩`}`

⟨*identifier*⟩ is the output stream identifier which should have been defined in an earlier call to `\open_pout`. ⟨*text*⟩ is what will be written to the file.

```
\def\pout_line#1#2{\immediate\write#1{#2}}
```

`\pout_lline` `\pout_llline` There relate to `\pout_line` as `\out_lline` and `\out_llline` to `\out_line`.

```
\edef\pout_lline#1#2{\noexpand\pout_line#1{\space\space\space#2}}
\edef\pout_llline#1#2{
   \noexpand\pout_line#1{\space\space\space\space\space\space#2}
}
```

\tempfileprefix Selecting the directory for temporary files.

\temp_prefix
```
\def\tempfileprefix#1{\def\temp_prefix{#1}}
\tempfileprefix{}
⟨/pkg⟩
```

8.3 Conditionals

8.3.1 The needs of fontinst

\then In order to write macros that expand out to nested \if-statements, I say:

\if_true
\if_false

\ifblah ... \then ... \else ... \fi

In order to match the \fi, \then has to be an \if.

```
⟨pkg | doc⟩\let\then=\iffalse
⟨*pkg⟩
\def\if_false{\iffalse}
\def\if_true{\iftrue}
```

\gobble_if It is sometimes best to skip both the then-part and the else-part of a conditional. \gobble_if does this by temporarily making \else mean \relax, thus preventing the expansion of \iffalse from removing anything less than all tokens up to and including the matching \fi.

```
\def\gobble_if{
   \begingroup
      \let\else=\x_relax
   \expandafter\endgroup
   \iffalse
}
```

\if_or \if_or ... \or_else ... \then gives the disjunction of two booleans.

\or_else
```
\def\if_or#1\or_else#2\then{
   #1\then
      \expandafter\if_true
   \else
      #2\then
         \expandafter\expandafter\expandafter\if_true
      \else
         \expandafter\expandafter\expandafter\if_false
      \fi
   \fi
}
```

\if_file_exists \if_file_exists checks to see if a file exits, using \openin.

```
\def\if_file_exists#1\then{
   \immediate\openin1=#1\x_relax
   \ifeof1\x_relax
      \immediate\closein1
      \expandafter\if_false
   \else
      \immediate\closein1
      \expandafter\if_true
   \fi
}
⟨/pkg⟩
```

8.3.2 The needs of fontdoc

In order to get a reasonable documentation of branches in an ETX or MTX file, the code in both branches must be typeset and the positions of the if, the else, and the fi must be clearly marked. There seems to be no reasonable way to achieve this if the standard names for the else and fi are used. Therefore the alternative names `\Else` and `\Fi` for `\else` and `\fi` have been introduced, to be used as in for example

```
\ifisint{monowidth}\then
  ⟨then-part⟩
\Else
  ⟨else-part⟩
\Fi
```

`\Else` `\Fi` By default, these are identical to their lowercase counterparts.

```
⟨*pkg | doc⟩
\let\Else=\else
\let\Fi=\fi
⟨/pkg | doc⟩
```

The interesting part begins if the ETX or MTX file gives the command `\showbranches`, since this means (i) that it wants the branches to be shown and (ii) that it complies to a simple rule about where to use `\Else` and `\Fi`. The rule is as follows: If an `if` statement is of fontinst form (it uses `\then`), then it must be terminated by `\Fi`, and a possible else in the statement must be an `\Else`. If an `if` statement is *not* of fontinst form (it has no `\then`), then it must be terminated by `\fi`, and a possible else in the statement must be an `\else`. Only the fontinst form `if` statements are affected by `\showbranches`.

`\showbranches` `\generic@if` The `\showbranches` command changes the definitions of `\generic@if`, `\then`, `\Else`, and `\Fi` so that the branches of `if` statements will be made visible. `\generic@if` is used to implement all the fontinst form `if` statements in fontdoc. It is called as

```
\generic@if{⟨description⟩}
```

and this will by default expand to `\iftrue`, but after executing `\showbranches` it will instead typeset

If ⟨*description*⟩ then

and do a bit of additional housekeeping.

```
⟨*doc⟩
\def\generic@if#1{\iftrue}

\newcommand\showbranches{%
    \let\generic@if=\branches@if
    \let\then=\x@relax
    \let\Else=\branches@else
    \let\Fi=\branches@fi
}
```

\saved@slot@number The \saved@slot@number is used for storing the value of \slot@number at the if until the \Else. Unlike \slot@number, it is always set locally.

```
\newcount\saved@slot@number
```

\branches@if \branches@else \branches@fi \branches@type \branches@par The macros \branches@if, \branches@else, and \branches@fi contain the definitions of \generic@if, \Else, and \Fi respectively that are used when branches are to be shown. Their basic task is to typeset the texts If #1 then, Else, and Fi in an appropriate style and with appropriate spacing around. A complicating matter is however that \slot@number must have the same value at the beginning of the *else* part as at the beginning of the *then* part. To achieve this, the value of \slot@numer is saved in \saved@slot@number at the *if* and copied back at the *else*. To make this work even with nested *if*s, both the *then* branch and the *else* branch are enclosed in groups and \saved@slot@number is assigned locally.

There are still some formating issues to solve, however. In some cases it works best to put the text of adjacent *if*, *else*, or *fi* in the same paragraph, and it must be possible to recognize those cases. This is done by looking at the macro \branches@type, which should expand to

- 0 if the preceeding item was an *if*,
- 1 if the preceeding item was an *else* preceeded by an *if*,
- 2 if the preceeding item was an *else* not preceeded by an *if*,
- 3 if the preceeding item was a *fi*,
- 4 if it was anything else.

\branches@par is used for resetting \branches@type to 3. It is an auto-resetting definition of \par.

```
\def\branches@if#1{%
   \ifnum \branches@type=\tw@
      \textit{ if #1 then}%
   \else
      \par\addvspace\medskipamount
      \noindent\textit{If #1 then}%
   \fi
   \saved@slot@number=\slot@number
   \begingroup
   \gdef\branches@type{0}%
   \let\par=\branches@par
}
\def\branches@else{%
   \endgroup
   \ifnum \branches@type=\z@
      \textit{ relax else}%
      \gdef\branches@type{1}%
   \else
      \par\addvspace\medskipamount
      \noindent\textit{Else}%
      \gdef\branches@type{2}%
   \fi
   \global\slot@number=\saved@slot@number
```

```
    \begingroup
    \let\par=\branches@par
}
\def\branches@fi{%
    \endgroup
    \ifnum \branches@type=\thr@@
       \textit{ fi}%
    \else
       \par\addvspace\medskipamount
       \noindent\textit{Fi}%
    \fi
    \gdef\branches@type{3}%
    \let\par=\branches@par
}
\gdef\branches@type{4}
\def\branches@par{%
    \@restorepar\par\gdef\branches@type{4}%
    \addvspace\medskipamount
}
⟨/doc⟩
```

8.3.3 Miscellanea

`\primitiveinput` If `\@@input` is defined, I'll assume it's the LaTeX version of the TeX `\input` primitive. I need this so that I can say `\expandafter\foo\primitiveinput`, which doesn't work with the LaTeX version of `\input`.

```
⟨*pkg⟩
\x_cs\ifx{@@input}\x_relax
    \let\primitiveinput=\input
\else
    \let\primitiveinput=\@@input
\fi
```

`\process_csep_list` The macro

`\process_csep_list{`⟨*pretext*⟩`}`⟨*comma-sep list*⟩`,\process_csep_list,`

executes ⟨*pretext*⟩`{`⟨*item*⟩`}` for every item in the ⟨*comma-sep list*⟩.

`\process_csep_list`

```
\def\process_csep_list#1#2,{
    \ifx\process_csep_list#2
       \expandafter\gobble_one
    \else
       \expandafter\identity_one
    \fi{
       #1{#2}
       \process_csep_list{#1}
    }
}
⟨/pkg⟩
```

`\print@csep@list` The macro call

`\print@csep@list{`⟨*format*⟩`}{`⟨*list*⟩`}`

prints the comma-separated list ⟨*list*⟩ in math mode. ⟨*format*⟩ can be used to set the style in which the list is printed, since it is executed in the beginning of the same group as in which the list is printed.

The formating is based on changing the `\mathcode` of comma to `"8000` so that the comma can be used as if it was an active character without actually having to be one.

```
⟨*doc⟩
\begingroup
   \catcode`\,=13
   \gdef\print@csep@list#1#2{%
      $%
      \ifnum "8000=\mathcode`\,\else
         \mathchardef\private@comma=\mathcode`\,%
         \mathcode`\,="8000\x@relax
      \fi
      #1%
      \let,=\active@comma
      #2%
      $%
   }
\endgroup
\def\active@comma{\private@comma \penalty\binoppenalty}
⟨/doc⟩
```

8.4 Real numbers

TEX's ⟨*number*⟩s are merely integers, but fontinst needs to store and calculate with numerical entities (most of which are lengths) that are intrinsically real numbers. Most of the time, fontinst does this by using a representation of real numbers by integers in which a real number x is represented by the integer that $1000x$ gets rounded to. This representation has proved sufficiently precise for most purposes.

There are however a few cases in which real numbers must be delt with in a more general fashion. Generic routines for this appear in this subsection.

`\make_factor` The `\make_factor` macro is meant to be used in the context

`\make_factor{`⟨*number*⟩`}`

where ⟨*number*⟩ can be any legal TEX number. It expands to the same number divided by 1000, with decimals, so that what it expands to matches the regular expression

`-?[0-9]+\.[0-9][0-9][0-9]`

More specifically, `\make_factor` has two tasks to perform: it takes care of the sign, so that its subsidary macros need only work with unsigned numbers, and it converts the ⟨*number*⟩ (which could well be a countdef or mathchardef token) to a sequence of digits.

```
⟨*pkg⟩
\def\make_factor#1{
   \ifnum #1<\z@
```

```
      -\expandafter\make_factor_i\expandafter{\number-#1}
   \else
      \expandafter\make_factor_i\expandafter{\number#1}
   \fi
}
```

`\make_factor_i` `\make_factor_i` distinguishes between the two main cases of numbers that in absolute value are less than 1000 and numbers that in absolute value are greater than or equal to 1000. In the former case, zeros need to be inserted. In the latter case, some number of digits need to be stepped over before the decimal point can be inserted.

```
\def\make_factor_i#1{
   \ifnum #1<\one_thousand
      \make_factor_ii{#1}
   \else
      \make_factor_iii #1
   \fi
   #1
}
```

`\make_factor_ii` `\make_factor_ii` inserts extra zeros, in the extent it is needed.

```
\def\make_factor_ii#1{
   0.
   \ifnum #1<\one_hundred
      0
      \ifnum #1<10~ 0\fi
   \fi
}
```

`\make_factor_iii` `\make_factor_iii` steps over one digit and checks if this is the correct position for the decimal point. Note that `#2#3` is exactly the digits in the number that has not been stepped over. Also note that comparision with `#2#3` would not work, since `#2` can be `0`.

```
\def\make_factor_iii#1#2#3\fi#4{
   \fi
   #4
   \ifnum 1#3<\one_thousand
      .
   \else
      \make_factor_iii#2#3
   \fi
}
```

⟨/pkg⟩

8.5 Error, warning, and info messages

The code in this subsection is an adaptation of code from the LaTeX kernel—more precisely from the source file `lterror.dtx`—and the same is true for some of the comments. As authors of that code are listed Johannes Braams, David Carlisle, Alan Jeffrey, Leslie Lamport, Frank Mittelbach, Chris Rowley, and Rainer Schöpf. The original source can be found in the file `lterror.dtx` in the `macros/latex/base` directory in any of the Comprehensive TeX Archive Network FTP sites.[4]

[4] As if anyone didn't already know that!

That should have fulfilled the conditions in the LPPL.

8.5.1 General commands

`\messagebreak` This command prints a new-line inside a message, followed by a continuation whose exact appearence may depend on the context. Normally this command is defined to be `\relax`, but inside messages it gets redefined to a linebreak (`^^J`) followed by the appropriate continuation.

```
⟨pkg⟩\let\messagebreak\x_relax
⟨doc⟩\let\messagebreak\space
```

`\generic_info` This takes two arguments: a continuation and a message, and sends the result to the log file.

```
⟨*pkg⟩
\def\generic_info#1#2{
   \begingroup
      \def\messagebreak{^^J#1}
      \immediate\write\m@ne{#2\on_line.}
   \endgroup
}
```

`\generic_warning` This takes two arguments: a continuation and a message, and sends the result to the screen.

```
\def\generic_warning#1#2{
   \begingroup
      \def\messagebreak{^^J#1}
      \immediate\write\closed_stream{^^J#2\on_line.^^J}
   \endgroup
}
```

`\generic_error` "This macro takes four arguments: a continuation, an error message, where to go for further information, and the help information. It displays the error message, and sets the error help (the result of typing `h` to the prompt), and does a horrible hack to turn the last context line (which by default is the only context line) into just three dots. This could be made more efficient."

Thus far the LaTeX comments, but the horrible hack it mensions has been removed since it just makes things look strange when `\errorcontextlines` is larger than one. The "where to go for further information" message is currently not used (it is commented out in `\fontinsterror` below), since there is no good place to refer to anyway. If the documentation is ever improved in this respect, it would of course be best to reinsert this message. Note however that unlike the case in LaTeX, this message should end with a `^^J`.

A complication is that TeX versions older than 3.141 have a bug which causes `^^J` to not force a linebreak in `\message` and `\errmessage` commands. So for these old TeX's we use a `\write` to produce the message, and then have an empty `\errmessage` command. This causes an extra line of the form

```
! .
```

to appear on the terminal, but if you do not like it, you can always upgrade your TeX!

Since I don't think many fontinst users use such old TEXs, I removed LATEX's test for TEX version at this point. The code for old TEXs is still there, but it lies in a docstrip module of its own.

First the 'standard case'.

```
⟨*!oldTeX⟩
\def\generic_error#1#2#3#4{
   \begingroup
      \immediate\write\closed_stream{}
      \def\messagebreak{^^J}
      \edef\a_macro{{#4}}
      \errhelp\a_macro
      \def\messagebreak{^^J#1}
      \errmessage{#2.^^J^^J#3
         Type~\space H~<return>~\space for~immediate~help
      }
   \endgroup
}
⟨/!oldTeX⟩
```

Secondly the version for old TEX's.

```
⟨*oldTeX⟩
\def\generic_error#1#2#3#4{%
   \begingroup
      \immediate\write\closed_stream{}
      \def\messagebreak{^^J}
      \edef\a_macro{{#4}}
      \errhelp\a_macro
      \def\messagebreak{^^J#1}
      \immediate\write\closed_stream{!~#2.^^J^^J#3^^J
         Type~\space H~<return>~\space for~immediate~help.
      }
      \errmessage{}
   \endgroup
}
⟨/oldTeX⟩
```

\fontinsterror
\fontinstwarning
\fontinstwarningnoline
\fontinstinfo

These commands are intended for giving a uniformed, and for the programmers hopefully simplified, form of error, warning, and info messages. The syntax is:

```
\fontinsterror{⟨subsystem⟩}{⟨error⟩}{⟨help⟩}
\fontinstwarning{⟨subsystem⟩}{⟨warning⟩}
\fontinstwarningnoline{⟨subsystem⟩}{⟨warning⟩}
\fontinstinfo{⟨subsystem⟩}{⟨info⟩}
```

A ⟨*subsystem*⟩ in this context is intended to be some piece of text which identifies some functionally clearly defined part of fontinst. Examples of ⟨*subsystem*⟩s could be `PL to MTX converter`, `Ligful (V)PL writer`, and `Latin family`.

The `\fontinsterror` command prints the ⟨*error*⟩ message, and presents the interactive prompt; if the user types `h`, then the ⟨*help*⟩ information is displayed. The `\fontinstwarning` command produces a warning but does not present the interactive prompt. The `\fontinstwarningnoline` command does the same, but doesn't print the input line number. The `\fontinstinfo` command writes the message to the `log` file. Within the messages, the command `\messagebreak` can be used to break a line and `\space` is a space, for example:

```
\def\foo{FOO}
\fontinstwarning{Hungarian}{
   Your~hovercraft~is~full~of~eels,\messagebreak
   and~\string\foo\space is~\foo}
```

produces:

```
Hungarian warning:
        Your hovercraft is full of eels,
        and \foo is FOO on input line 54.
```

```
\def\fontinsterror#1#2#3{
   \generic_error{
      \four_spaces\four_spaces
   }{
      #1~error:\messagebreak #2
   }{
%       See~the~#1~package~documentation~for~explanation.^^J
   }{#3}
}
\def\fontinstwarning#1#2{
   \generic_warning{
      \four_spaces\four_spaces
   }{
      #1~warning:\messagebreak #2
   }
}
\def\fontinstwarningnoline#1#2{
   \fontinstwarning{#1}{#2\gobble_one}
}
\def\fontinstinfo#1#2{
   \generic_info{
      \four_spaces\four_spaces
   }{
      #1~info:\messagebreak #2
   }
}
⟨/pkg⟩
⟨*doc⟩
\def\fontinsterror#1#2#3{%
   \Bheading{Error} observed by #1:%
   \begin{quote}#2.\end{quote}%
}
\def\fontinstwarning#1#2{
   \Bheading{Warning} from #1:%
   \begin{quote}#2.\end{quote}%
}
\let\fontinstwarningnoline=\fontinstwarning
\def\fontinstinfo#1#2{
   \Bheading{Info} from #1:%
```

```
    \begin{quote}#2.\end{quote}%
}
```
⟨/doc⟩

`\on_line` The message ‘ on input line n’. LaTeX has special code for TeX 2, but since fontinst has assumed the existence of the `\errorcontextlines` parameter since v 1.500, the removal of that code shouldn't cause problems for anyone who wasn't already having related problems.

⟨∗pkg⟩
```
\def\on_line{~on~input~line~\the\inputlineno}
```

`\four_spaces` Four spaces. Using `\edef` (rather than `\def` as in LaTeX) to save some macro expansions.

```
\edef\four_spaces{\space\space\space\space}
```

8.5.2 Specific errors

`\error_help_a` `\error_help_c` `\error_help_d` The more common error help messages. They are called `\@eha`, `\@ehc`, and `\@ehd` in LaTeX. `\@ehb` is of no use for fontinst, so it has been omitted.

```
\gdef\error_help_a{
  Your~command~was~ignored.\messagebreak
  Type~\space I~<command>~<return>~\space to~replace~it~
  with~another~command,\messagebreak
  or~\space <return>~\space to~continue~without~it.}
\gdef\error_help_c{
  Try~typing~\space <return>~
  \space to~proceed.\messagebreak
  If~that~doesn't~work,~type~\space X~<return>~\space to~quit.}
\gdef\error_help_d{
  You're~in~trouble~here.~\space\error_help_c}
```

⟨/pkg⟩

File b
fimain.dtx

9 General commands

9.1 Setting variables

⟨*int*⟩, ⟨*str*⟩, and ⟨*dim*⟩ below can be any strings. ⟨*command*⟩ can be any control sequence.

\setint
\setstr
\setdim
\setcommand

The macros:

\setint{⟨*int*⟩}{⟨*integer expression*⟩}
\setstr{⟨*str*⟩}{⟨*string*⟩}
\setdim{⟨*dim*⟩}{⟨*dimension*⟩}
\setcommand⟨*command*⟩⟨*definition*⟩

define new macros \i-⟨*int*⟩, \s-⟨*str*⟩, \d-⟨*dim*⟩ or ⟨*command*⟩.

\resetint
\resetstr
\resetdim
\resetcommand

The macros:

\resetint{⟨*int*⟩}{⟨*integer expression*⟩}
\resetstr{⟨*str*⟩}{⟨*string*⟩}
\resetdim{⟨*dim*⟩}{⟨*dimension*⟩}
\resetcommand⟨*command*⟩⟨*definition*⟩

redefine the macros \i-⟨*int*⟩, \s-⟨*str*⟩, \d-⟨*dim*⟩ or ⟨*command*⟩.

\int
\str
\dim

The macros:

\int{⟨*int*⟩}
\str{⟨*str*⟩}
\dim{⟨*dim*⟩}
⟨*command*⟩

return the values of \i-⟨*int*⟩, \s-⟨*str*⟩, \d-⟨*dim*⟩, or ⟨*command*⟩.

\strint

The macro

\strint{⟨*int*⟩}

returns the value of \i-⟨*int*⟩ as a string.

\ifisint
\ifisstr
\ifisdim
\ifiscommand

The macros:

\ifisint{⟨*int*⟩}\then
\ifisstr{⟨*str*⟩}\then
\ifisdim{⟨*dim*⟩}\then
\ifiscommand⟨*command*⟩\then

return \if_true if \i-⟨*int*⟩, \s-⟨*str*⟩, \d-⟨*dim*⟩, or ⟨*command*⟩ have been defined, and \if_false otherwise.

\unsetint
\unsetstr
\unsetdim
\unsetcommand

The macros:

\unsetint{⟨*int*⟩}
\unsetstr{⟨*str*⟩}
\unsetdim{⟨*dim*⟩}
\unsetcommand⟨*command*⟩

undefine \i-⟨*int*⟩, \s-⟨*str*⟩, \d-⟨*dim*⟩, or ⟨*command*⟩.

\x_setint
\x_resetint
\x_setstr

The macros \x_setint, \x_resetint, and \x_setstr are "private" versions of \setint, \resetint, and \setstr respectively. They have been included to reduce the problems in case a user turns off one of these commands and forgets to turn it on again.

Integers are kept as \mathchardefs if possible; a comparision of doing this versus not doing this appears in Subsection E.1.

\x_setint
\setint
\x_setstr
\setstr
\setdim
\setcommand

```
⟨*pkg | misc⟩
\def\x_setint#1#2{
   \x_cs\ifx{i-#1}\x_relax
      \x_resetint{#1}{#2}
   \fi
}
\let\setint=\x_setint
\def\x_setstr#1#2{\x_cs\ifx{s-#1}\x_relax
   \x_cs\edef{s-#1}{#2}\fi}
\let\setstr=\x_setstr
\def\setdim#1#2{\x_cs\ifx{d-#1}\x_relax
   \a_dimen=#2\x_relax\x_cs\edef{d-#1}{\the\a_dimen}\fi}
\def\setcommand#1{\ifx#1\undefined_command
   \expandafter\def\expandafter#1\else
   \expandafter\def\expandafter\a_command\fi}
⟨/pkg | misc⟩
```

In fontdoc, \setint, \setstr, and \setdim print headings. There is no need for the private forms \x_setint and \x_setstr. \setcommand, finally, does the same thing as in fontinst.

```
⟨*doc⟩
\def\setint#1#2{\Bheading{Default} #1 = $\expression0{#2}$}
\def\setstr#1#2{\Bheading{Default} #1 = #2}
\def\setdim#1#2{\a@dimen=#2\relax\Bheading{Default} #1 = \the\a@dimen}
\def\setcommand#1{\ifx#1\undefined@command
   \expandafter\gdef\expandafter#1\else
   \expandafter\gdef\expandafter\a@command\fi}
⟨/doc⟩
```

\x_resetint
\resetint
\resetstr
\resetdim
\resetcommand

```
⟨*pkg | misc⟩
\def\x_resetint#1#2{
   \eval_expr{#2}
   \ifnum\result<\max_mathchardef
      \ifnum 0>\result
         \x_cs\edef{i-#1}{\the\result}
      \else
         \x_cs\mathchardef{i-#1}=\result
      \fi
   \else
      \x_cs\edef{i-#1}{\the\result}
   \fi
}
\let\resetint=\x_resetint
\def\resetstr#1#2{\x_cs\edef{s-#1}{#2}}
```

```
\def\resetdim#1#2{\a_dimen=#2\x_cs\edef{d-#1}{\the\a_dimen}}
\def\resetcommand#1{\def#1}
⟨/pkg | misc⟩
```

In fontdoc, `\resetint`, `\resetstr`, and `\resetdim` print headings. There is no need for the private form `\x_resetint`. `\resetcommand`, finally, does the same thing as in fontinst.

```
⟨*doc⟩
\def\resetint#1#2{\Bheading{Value} #1 = $\expression0{#2}$}
\def\resetstr#1#2{\Bheading{Value} #1 = #2}
\def\resetdim#1#2{\a@dimen=#2\relax\Bheading{Value} #1 = \the\a@dimen}
\def\resetcommand#1{\gdef#1}
⟨/doc⟩
```

`\int` `\str` `\dim` `\strint`

```
⟨*pkg | misc⟩
\def\int#1{\csname~i-#1\endcsname}
\def\str#1{\csname~s-#1\endcsname}
\def\dim#1{\csname~d-#1\endcsname}
\def\strint#1{\expandafter\identity_one\expandafter{\number\int{#1}}}
⟨/pkg | misc⟩
```

Not in doc: `\str` Not in doc: `\dim` Not in doc: `\strint`

```
⟨*doc⟩
\def\int#1{\typeset@integer{#1}}
```

`\typeset@integer` `\typeset@string` `\typeset@dimen`

These macros take the name of an integer/string/dimen as argument and typesets that name with the correct formating. They should work equally well in horizontal, math, and vertical mode.

```
\def\typeset@integer#1{\x@relax \ifmmode {\fam0 #1}\else $\fam0 #1$\fi}
\def\typeset@string#1{\x@relax
   \ifmmode \hbox{\normalfont#1}\else \textnormal{#1}\fi
}
\let\typeset@dimen=\typeset@string
⟨/doc⟩
```

`\ifisint` `\ifisstr` `\ifisdim` `\ifiscommand`

```
⟨*pkg | misc⟩
\def\ifisint#1\then{\x_cs\ifx{i-#1}\x_relax\expandafter\if_false
   \else\expandafter\if_true\fi}
\def\ifisstr#1\then{\x_cs\ifx{s-#1}\x_relax\expandafter\if_false
   \else\expandafter\if_true\fi}
\def\ifisdim#1\then{\x_cs\ifx{d-#1}\x_relax\expandafter\if_false
   \else\expandafter\if_true\fi}
\def\ifiscommand#1\then{\ifx#1\undefined_command\expandafter\if_false
   \else\expandafter\if_true\fi}
⟨/pkg | misc⟩
```

In fontdoc, all conditionals are handled through `\generic@if`, which by default expands to `\iftrue`.

```
⟨*doc⟩
\def\ifisint#1\then{\generic@if{integer \typeset@integer{#1} set}}
\def\ifisstr#1\then{\generic@if{string \typeset@string{#1} set}}
\def\ifisdim#1\then{\generic@if{dimension \typeset@dimen{#1} set}}
\def\ifiscommand#1\then{%
```

```
    \generic@if{command \normalfont{\ttfamily\string#1} set}%
}
```
⟨/doc⟩

\unsetint
\unsetstr
\unsetdim
\unsetcommand

⟨∗pkg | misc⟩
```
\def\unsetint#1{\x_cs\let{i-#1}\x_relax}
\def\unsetstr#1{\x_cs\let{s-#1}\x_relax}
\def\unsetdim#1{\x_cs\let{d-#1}\x_relax}
\def\unsetcommand#1{\let#1=\undefined_command}
```
⟨/pkg | misc⟩

Not in doc: \unsetint
Not in doc: \unsetstr
Not in doc: \unsetdim
Not in doc: \unsetcommand

⟨∗doc⟩
⟨/doc⟩

\offcommand
\oncommand

LH 1999/02: The calls

\offcommand⟨*command*⟩
\oncommand⟨*command*⟩

can be used to turn a command off (make it simply absorb its arguments but expand to nothing) or on (return it to normal) respectively. Their primary use is for ignoring some of the commands in `.mtx` files that fontinst generates from `.afm`, `.pl`, or other `.mtx` files.

\saved-COMMAND

The normal definition of a command that have been turned off is saved as the control sequence \saved-⟨*command*⟩. If the syntax of a command is tricky—not all arguments are read by the base command or its parameter text contains tokens that are not parameters—or if simply making it do nothing is not the expected 'off' behaviour, then the automatic off-turning may not work. In such cases a handtooled off definition of the command ⟨*command*⟩ may be provided as the control sequence \off-⟨*command*⟩.

\off-COMMAND

Nothing happens if \offcommand is called for a command that is not on. Nothing happens if \oncommand is called for a command that is not off.

⟨∗pkg | misc⟩
```
\def\offcommand#1{
   \x_cs\ifx{saved-\string#1}\x_relax
      \x_cs\let{saved-\string#1}#1
      \x_cs\ifx{off-\string#1}\x_relax
         \generate_off_command{#1}
      \else
         \expandafter\let \expandafter#1
            \csname off-\string#1\endcsname
      \fi
   \fi
}
\def\oncommand#1{
   \x_cs\ifx{saved-\string#1}\x_relax \else
      \expandafter\let \expandafter#1
            \csname saved-\string#1\endcsname
      \x_cs\let{saved-\string#1}\x_relax
   \fi
}
```

\generate_off_command
\count_hashes
\gobble_to_xrelax

\generate_off_command⟨*command*⟩ converts ⟨*command*⟩ to an "off" version of itself by counting the number of arguments and defining it to gobble that many arguments. It cannot cope with commands whose parameter texts are not simply of the type

#1#2...#n

```
\def\generate_off_command#1{
   \a_count=0
   \let\next=\count_hashes
   \expandafter\next\meaning#1~->\x_relax
   \b_count=0
   \a_toks={}
   \loop \ifnum \b_count<\a_count
      \advance \b_count 1
      \a_toks=\expandafter{\the\expandafter\a_toks \expandafter####
         \the\b_count}
   \repeat
   \expandafter\def \expandafter#1 \the\a_toks {}
}
\def\count_hashes#1#2{
   \if \hash_char#1
      \advance \a_count 1
   \else
      \if -#1
         \if >#2
            \let\next=\gobble_to_xrelax
      \fi\fi
   \fi
   \next#2
}
\def\gobble_to_xrelax#1\x_relax{}
```

Not in doc: \offcommand
Not in doc: \oncommand

⟨/pkg | misc⟩

⟨*doc⟩
⟨/doc⟩

9.2 For loops

\for
\endfor

LH 1999/02: The command sequence

\for(⟨*name*⟩){⟨*start*⟩}{⟨*stop*⟩}{⟨*step*⟩}
⟨*body code*⟩
\endfor(⟨*name*⟩)

will cause the ⟨*body code*⟩ to be repeated some number of times. How many depends on the values of ⟨*start*⟩, ⟨*stop*⟩, and ⟨*step*⟩, which are integer expressions.

As a precaution, the ⟨*body code*⟩ is not allowed to contain any empty lines (\par tokens). If you want to have the visual separation (for sakes of legibility or otherwise), put a % somewhere on the line—that makes it nonempty.

⟨*name*⟩ is used as the name of an integer variable. This variable gets reset to the value of ⟨*start*⟩ before the first repetition of ⟨*body code*⟩. After each repetition but the last, it is incremented by ⟨*step*⟩. ⟨*body code*⟩ gets repeated if the value of ⟨*name*⟩ has not gotten past that of ⟨*stop*⟩. To get past means to be bigger if

⟨*step*⟩ is positive and to be smaller if ⟨*step*⟩ is negative. In the case that ⟨*step*⟩ is zero, the entire construction above will be equivalent to

\resetint{⟨*name*⟩}{⟨*start*⟩}
⟨*body code*⟩

\for ... \endfor constructions can be nested. ⟨*name*⟩ is used by \for to identify its matching \endfor, so they need to be identical in \for and \endfor.

```
⟨*pkg | misc⟩
\def\for(#1)#2#3#4{
   \eval_expr_to\a_count{#2}
   \x_resetint{#1}{\a_count}
   \eval_expr{#4}
   \ifnum 0=\result \else
      \c_count=\result
      \eval_expr_to\b_count{#3}
      \x_cs\edef{for-#1}{
         \the\c_count \x_relax
         \noexpand\ifnum \gobble_one\fi
         \the\b_count \ifnum 0>\c_count > \else < \fi
      }
      \def\next##1##2##3\endfor(#1){##2\for_i{##1}{##3}}
      \next{#1}
   \fi
}
```

\for-NAME The macro \for-⟨*name*⟩ will contain

⟨*step*⟩\x_relax\ifnum⟨*stop*⟩⟨*rel*⟩

⟨*step*⟩ is the value of the ⟨*step*⟩ parameter of \for, computed when the loop was entered and now expressed in digits. ⟨*stop*⟩ is likewise for the ⟨*stop*⟩ parameter. ⟨*rel*⟩ is > or <, depending on whether ⟨*step*⟩ is positive or negative respectively. The reason for this curious definition will be appearent in the light of the definition of \for_ii.

\for_i is expanded in the context

\for_i{⟨*name*⟩}{⟨*body code*⟩}

Also remember, when reading the definition below, that \ifnum keeps on expanding tokens until it has found a

⟨*number*⟩⟨*relation*⟩⟨*number*⟩

structure. It is therefore possible to nest \ifnums like this!

```
\def\for_i#1#2{
   \x_cs\def{body-#1}{#2}
   \ifnum \b_count \ifnum0>\c_count >\else<\fi \a_count
      \expandafter\gobble_two
   \else
      \csname body-#1 \expandafter\endcsname
   \fi
   \for_ii{#1}
}
```

\body-NAME

The macro \body-⟨*name*⟩ expands to the ⟨*body code*⟩.
\for_ii executes the following code:

```
\a_count=\int{⟨name⟩}
\advance \a_count ⟨step⟩\x_relax
\ifnum ⟨stop⟩⟨rel⟩\a_count
   \expandafter\gobble_two
\else
   \resetint{⟨name⟩}\a_count
   \csname body-⟨name⟩ \expandafter\endcsname
\fi
\for_ii{⟨name⟩}
```

⟨*step*⟩, ⟨*stop*⟩, and ⟨*rel*⟩ are in \for-⟨*name*⟩, and since there only are two other tokens between ⟨*step*⟩ and ⟨*rel*⟩ in the above, one might as well include them in \for-⟨*name*⟩ as well. Doing that requires that a matching hole—that will be filled in by \for-⟨*name*⟩—is made in the definition of \for_ii and that is the reason for its somewhat curious definition.

```
\def\for_ii#1{
   \a_count=\int{#1}
   \advance \a_count \csname for-#1\endcsname \a_count
      \expandafter\gobble_two
   \else
      \x_resetint{#1}\a_count
      \csname body-#1 \expandafter\endcsname
   \fi
   \for_ii{#1}
}
```

\endfor just gobbles its argument, so that the ⟨*step*⟩ = 0 case will work right.

```
\def\endfor(#1){}
```

⟨/pkg | misc⟩

Not in doc: \for

Not in doc: \endfor

⟨∗doc⟩
⟨/doc⟩

\foreach
\foreach_i

The command sequence

```
\foreach(⟨name⟩){⟨csep-list⟩}
  ⟨body code⟩
\endfor(⟨name⟩)
```

will cause the ⟨*body code*⟩ to be repeated one time for each item in the ⟨*csep-list*⟩. ⟨*csep-list*⟩ is a comma-separated list of strings.

As a precaution, the ⟨*body code*⟩ is not allowed to contain any empty lines (\par tokens). If you want to have the visual separation (for sakes of legibility or otherwise), put a % somewhere on the line—that makes it nonempty.

⟨*name*⟩ is used as the name of a string variable. Before each repetition of the ⟨*body code*⟩, this variable will get reset to the next item in the ⟨*csep-list*⟩.

\for... \endfor constructions can be nested. ⟨*name*⟩ is used by \foreach to identify its matching \endfor, so they need to be identical in \foreach and \endfor.

Not in doc: \foreach

```
⟨*pkg | misc⟩
\def\foreach(#1)#2{
   \def\next##1\endfor(#1){
      \x_cs\def{body-#1}{##1}
      \process_csep_list{\foreach_i{#1}}#2,\process_csep_list,
   }
   \next
}
\def\foreach_i#1#2{
   \resetstr{#1}{#2}
   \csname body-#1\endcsname
}
⟨/pkg | misc⟩
```

9.3 Integer expressions

\eval_expr
\eval_expr_to
\g_eval_expr_to

The macro

\eval_expr{⟨*integer expression*⟩}

globally assigns \result to the value of ⟨*integer expression*⟩, and changes the value of no other counters.

The macro

\eval_expr_to{⟨*integer variable*⟩}{⟨*integer expression*⟩}

locally assigns the value of ⟨*integer expression*⟩ to ⟨*integer variable*⟩ (which is an integer variable as defined in [1]). \g_eval_expr_to does the same globally.

```
⟨*pkg | misc⟩
\newcount\result
\def\eval_expr#1{\global\result=#1\x_relax}
\def\eval_expr_to#1#2{\eval_expr{#2}#1=\result}
\def\g_eval_expr_to#1#2{\eval_expr{#2}\global#1=\result}
```

\rounded_thousandths

LH 1999/03/06: The \rounded_thousandths macro divides \result by 1000 and rounds the result to the nearest integer. This is different from

\global\divide \result \one_thousand

since the latter always rounds positive numbers downwards and negative numbers upwards.

```
\def\rounded_thousandths{
   \global\divide \result \five_hundred
   \ifodd \result
      \global\advance \result by \ifnum 0>\result - \fi 1
   \fi
   \global\divide \result \tw@
}
```

\neg
\add
\sub
\mul
\div
\max
\min

These macros return an integer expression:

\neg{⟨*integer expression*⟩}
\add{⟨*integer expression*⟩}{⟨*integer expression*⟩}
\sub{⟨*integer expression*⟩}{⟨*integer expression*⟩}

\mul{⟨*integer expression*⟩}{⟨*integer expression*⟩}
\div{⟨*integer expression*⟩}{⟨*integer expression*⟩}
\max{⟨*integer expression*⟩}{⟨*integer expression*⟩}
\min{⟨*integer expression*⟩}{⟨*integer expression*⟩}
\scale{⟨*integer expression*⟩}{⟨*integer expression*⟩}

```
\def\neg#1{#1\global\result=-\result}
\def\add#1#2{#1{\a_count=\result\eval_expr{#2}
   \global\advance\result by \a_count}}
\def\sub#1#2{#1{\a_count=\result\eval_expr{#2}
   \advance\a_count by -\result
   \global\result=\a_count}}
\def\mul#1#2{#1{\a_count=\result\eval_expr{#2}
   \global\multiply\result by \a_count}}
\def\div#1#2{#1{\a_count=\result\eval_expr{#2}
   \divide\a_count by \result
   \global\result=\a_count}}
\def\max#1#2{#1{\a_count=\result\eval_expr{#2}
   \ifnum\a_count>\result \global\result=\a_count \fi}}
\def\min#1#2{#1{\a_count=\result\eval_expr{#2}
   \ifnum\a_count<\result \global\result=\a_count \fi}}
⟨/pkg | misc⟩

⟨*doc⟩
\def\neg#1{\priority8{-\expression6{#1}}}
\def\add#1#2{\priority2{\expression2{#1}+\expression2{#2}}}
\def\sub#1#2{\priority2{\expression2{#1}-\expression3{#2}}}
\def\mul#1#2{\priority4{\expression4{#1}\times\expression4{#2}}}
\def\div#1#2{\priority4{\expression4{#1}/\expression5{#2}}}
\def\max#1#2{\priority6{\expression6{#1}\sqcup\expression6{#2}}}
\def\min#1#2{\priority7{\expression7{#1}\sqcap\expression7{#2}}}
⟨/doc⟩
```

\scale
\grabchars@
\grabchars@i
\grabchars@ii
\grabchars@iii
\scale-500X

The `\scale` macro performs the operation of multiplying the operands and dividing that result by one thousand.

```
⟨*pkg | misc⟩
\def\scale#1#2{#1{\a_count=\result\eval_expr{#2}
   \global\multiply\result by \a_count
   \rounded_thousandths}}
⟨/pkg | misc⟩
```

`\scale` is often used with a fixed scaling factor, such as for example 500, as the second argument. In those cases it often makes more sense to typeset the expression as for example a fraction times the first argument than to typeset it as `\div{\mul{#1}{#2}}{1000}` would. Such special cases can be specified by defining a control sequence `\scale-`⟨*factor*⟩, where ⟨*factor*⟩ is the first four characters of the `#2` integer expression (all control sequences are `\string`ed), with `X`s added at the end if the expression is shorter than four characters.

\scale-FACTOR

For the moment, the only scaling factor with such a special definition is 500. This scaling factor gets typeset as $\frac{1}{2}$.

```
⟨*doc⟩
\def\scale#1#2{%
   \x@cs\ifx{scale-\grabchars@#2XXXX\@nil}\x@relax
      \div{\mul{#1}{#2}}{1000}%
```

```
    \else
       \csname scale-\grabchars@#2XXXX\@nil\endcsname{#1}{#2}%
    \fi
}
\def\grabchars@{\expandafter\grabchars@i\string}
\def\grabchars@i#1{#1\expandafter\grabchars@ii\string}
\def\grabchars@ii#1{#1\expandafter\grabchars@iii\string}
\def\grabchars@iii#1{#1\expandafter\@car\string}
\x@cs\def{scale-500X}#1#2{\priority4{\frac{1}{2}\expression4{#1}}}
⟨/doc⟩
```

`\expression` `\priority` `\identity` `\bracket`

fontdoc uses `\expression` and `\priority` for typsetting nested arithmetic expressions. `\expression` is called as

`\expression{`⟨*priority*⟩`}{`⟨*expression*⟩`}`

where ⟨*priority*⟩ is the minimum priority the outermost operation in ⟨*expression*⟩ must have if ⟨*expression*⟩ is to be read correctly without bracketing. `\priority` is called as

`\priority{`⟨*priority*⟩`}{`⟨*expression*⟩`}`

where ⟨*priority*⟩ is the priority the outermost operation in ⟨*expression*⟩. Its object is to bracket ⟨*expression*⟩ if that is necessary.

```
⟨*doc⟩
\def\expression#1#2{\a@count=#1\relax#2}
\def\priority#1#2{%
    \ifnum #1<\a@count
       \let\next=\bracket
    \else
       \let\next=\identity
    \fi
    \next{#2}}
\def\identity#1{#1}
\def\bracket#1{(#1)}
⟨/doc⟩
```

`\ifnumber`

LH 1999/02: The call

`\ifnumber{`⟨*integer expression*⟩`}`⟨*rel*⟩`{`⟨*integer expression*⟩`}\then`

can be used to compare the two integer expressions. ⟨*rel*⟩ may be `<`, `=`, or `>`.

```
⟨*pkg | misc⟩
\def\ifnumber#1#2#3\then{
    \eval_expr_to\a_count{#1}
    \eval_expr{#3}
    \ifnum \a_count#2\result
       \expandafter\if_true
    \else
       \expandafter\if_false
    \fi
}
⟨/pkg | misc⟩
```

Like the other conditionals, `\ifnumber` is treated as being true by fontdoc.

```
⟨*doc⟩
\def\ifnumber#1#2#3\then{%
   \generic@if{$ \expression0{#1} #2 \expression0{#3} $}%
}
⟨/doc⟩
```

9.4 Comments

`\comment` In fontinst, `\comment` simply gobbles its argument.

```
⟨*pkg⟩
\let\comment=\gobble_one
⟨/pkg⟩
```

In fontdoc, `\comment` starts a new text paragraph and leaves the argument to be typeset.

```
⟨*doc⟩
\def\comment{\par\noindent}
⟨/doc⟩
```

`\begincomment` `\endcomment` LH 1999/02: Since `\comment` cannot be used for a comment longer than one paragraph, we also provide the means of introducing longer comments, by writing

`\begincomment` ⟨*any amount of text*⟩ `\endcomment`

The names are admittedly not nice from a LaTeX point of view, but it is hardly worth the cost to implement some kind of environment processing in fontinst, just for the sake of this command.

```
⟨pkg⟩\let\begincomment=\iffalse
⟨doc⟩\let\begincomment=\iftrue
⟨pkg | doc⟩\let\endcomment=\fi
```

10 Encoding files

`\inputetx` The macro

`\inputetx{`⟨*filename*⟩`}`

inputs ⟨*filename*⟩`.etx`, ignoring anything between `\relax` and `\encoding`, and anything after `\endencoding`.

The file name is transformed to lowercase before opening.

`\inputetx` `\encoding` `\endencoding`

```
⟨*pkg⟩
\def\inputetx#1{
   \edef\lowercase_file{\lowercase{
     \edef\noexpand\lowercase_file{#1}}}
   \lowercase_file
   \slot_number=0
   \def\relax{\let\relax=\x_relax\iffalse}
   \let\encoding=\fi
   \primitiveinput \lowercase_file.etx\x_relax
   \let\relax=\x_relax
```

```
}
\let\encoding=\relax
\outer\def\endencoding{\endinput}
⟨/pkg⟩
```

Things are a bit more complicated in fontdoc, since the `\relax` ... `\encoding` ... `\endencoding` markup must be able to work in two different ways. In the main file only `\encoding` actually does anything—it sets `\slot@number` to zero. In a file that is being `\inputetx`ed, they must work as with fontinst: Everything between `\relax` and `\encoding`, and everything after `\endencoding` must be ignored.

```
⟨*doc⟩
\def\inputetx#1{%
   \begingroup
      \edef\lowercase@file{\lowercase{%
         \edef\noexpand\lowercase@file{#1}%
      }}%
      \lowercase@file
      \global\slot@number=0%
      \def\relax{\let\relax=\x@relax\iffalse}%
      \let\encoding=\fi
      \outer \expandafter\def \csname endencoding\endcsname
         {\endinput}%
      \expandafter\input \lowercase@file.etx\x@relax
      \let\relax=\x@relax
   \endgroup
}
\def\encoding{\begingroup\global\slot@number=0}
\outer\def\endencoding{\endgroup}
⟨/doc⟩
```

`\setslot` `\endsetslot` `\slot_name` `\slot@name`

`\setslot{`⟨*name*⟩`}` ⟨*slot commands*⟩ `\endsetslot`

In fontinst, this sets `\slot_name` to ⟨*name*⟩ and calls `\do_slot` at the beginning of the slot and it calls `\end_do_slot` and increments `\slot_number` by one at the end. By default, `\do_slot` and `\end_do_slot` do nothing, but this is over-ridden later.

```
⟨*pkg⟩
\def\setslot#1{\edef\slot_name{#1}\do_slot}
\def\endsetslot{\end_do_slot\advance\slot_number by 1\x_relax}
⟨/pkg⟩
```

In fontdoc, the same code sets `\slot@name` to ⟨*name*⟩, prints an `\Aheading` heading for the slot which show the name and number of the slot, and prints the current automatic slot comment (if one has been set) at the beginning of the slot. At the end of the slot, it simply increments `\slot@number` by one.

```
⟨*doc⟩
\def\setslot#1{\def\slot@name{#1}%
   \Aheading{Slot \the\slot@number\space `#1'}%
   \ifslot@comment@ \comment{\slot@comment}\fi}
\def\endsetslot{\global\advance\slot@number by 1\relax}
⟨/doc⟩
```

`\do_slot` `\end_do_slot`

```
⟨*pkg⟩
```

```
326 \let\do_slot\empty_command
327 \let\do_new_slot\empty_command
328 \let\end_do_slot\empty_command
```

\nextslot
\skipslots
\slot_number
\slot@number

`\nextslot{`⟨*integer expression*⟩`}`
`\skipslots{`⟨*integer expression*⟩`}`

In fontinst, `\nextslot` sets the `\slot_number` and `\skipslots` addvances the `\slot_number`.

```
329 \newcount\slot_number
330 \def\nextslot#1{\eval_expr_to\slot_number{#1}}
331 \def\skipslots#1{\eval_expr{#1} \advance\slot_number by \result}
332 ⟨/pkg⟩
```

The commands do the same in fontdoc as in fontinst, although they do it to `\slot@number` instead of `\slot_number`.

```
333 ⟨*doc⟩
334 \newcount\slot@number
335 \def\nextslot#1{\global\slot@number=#1\relax}
336 \def\skipslots#1{\global\advance\slot@number by #1\relax}
337 ⟨/doc⟩
```

A **What should the arguments to `\nextslot` and `\skipslots` be?** At least since v 1.335 the comments in the source has said it was an integer expression, but the implementation was for a TEX number. Alan's v 1.5 manual and Rowland's v 1.8 manual both say the arguments must be numbers. Allowing arbitrary integer expressions with fontinst is trivial, the above implementation copes with that, but fontdoc gets in trouble, so what should we do about it? /LH

AFAIK, all instances of `\nextslot` or `\skipslots` appearing in present `*.etx` files are explicit numbers, no fancy constructs. /UV

\setleftboundary
\endsetleftboundary
\do_boundary

`\setleftboundary{`⟨*glyph*⟩`}` ⟨*slot commands*⟩ `\endsetleftboundary`

These macros are like `\setslot` and `\endsetslot`, but they merely set the left boundary ligkern program, they do not cause any `CHARACTER` property list to be written. Thus the only metric information connected to the ⟨*glyph*⟩ argument that is ever used is the kerns with this glyph on the left.

`\do_boundary` and `\endsetleftboundary` are initally `\relax`, but are later redefined.

```
338 ⟨*pkg⟩
339 \def\setleftboundary#1{\edef\slot_name{#1}\do_boundary}
340 \let\endsetleftboundary\x_relax
341 \let\do_boundary\x_relax
342 ⟨/pkg⟩
343 ⟨*doc⟩
344 \def\setleftboundary#1{\def\slot@name{#1}%
345    \Aheading{Left boundary `#1'}%
346 }
347 \let\endsetleftboundary=\x@relax
348 ⟨/doc⟩
```

\setrightboundary

`\setrightboundary{`⟨*glyph*⟩`}`

The `\setrightboundary` macro should be used to set which slot in the font is used as right boundary marker if that slot is empty. It advances `\slot_number`

just like a `\setslot` ... `\endsetslot` pair, but since the slot will be left empty, there is no need for any ⟨*slot commands*⟩, and hence there is no need for a closing `\endset`... command either.

If the right boundary marker slot is not to be left empty (often unavoidable), then one should use the slot command `\makerightboundary` instead.

```
⟨*pkg⟩
\def\setrightboundary#1{
   \makerightboundary{#1}
   \advance \slot_number 1\x_relax
}
⟨/pkg⟩
⟨*doc⟩
\def\setrightboundary#1{%
   \Aheading{Right boundary slot \the\slot@number\space ‘#1’}%
   \global\advance\slot@number by 1\relax
}
⟨/doc⟩
```

`\ligature`
`\nextlarger`
`\varchar`
`\endvarchar`
`\usedas`
`\makerightboundary`

The ⟨*slot commands*⟩ are:

```
\ligature{⟨lig⟩}{⟨glyph⟩}{⟨glyph⟩}
\nextlarger{⟨glyph⟩}
\varchar ⟨varchar commands⟩ \endvarchar
\usedas{⟨command⟩}{⟨type⟩}
\makerightboundary{⟨glyph⟩}
```

By default, these do nothing in fontinst, but those defaults are overridden later.[5] In fontdoc, they generate descriptive headings.

`\ligature`

```
⟨pkg⟩\let\ligature=\gobble_three
⟨*doc⟩
\def\ligature#1{\Bheading{Ligature} \csname doc-#1\endcsname}
```

Symbolic typesetting of `\ligature` programs requires special processing.

```
\x@cs\def{doc-LIG}#1#2{%
  $\typeset@glyph{\slot@name} * \typeset@glyph{#1} \rightarrow
  \typeset@glyph{#2}$}
\x@cs\def{doc-/LIG}#1#2{%
  $\typeset@glyph{\slot@name} * \typeset@glyph{#1} \rightarrow
  \typeset@glyph{\slot@name} * \typeset@glyph{#2}$}
\x@cs\def{doc-LIG/}#1#2{%
  $\typeset@glyph{\slot@name} * \typeset@glyph{#1} \rightarrow
  \typeset@glyph{#2} * \typeset@glyph{#1}$}
\x@cs\def{doc-/LIG/}#1#2{%
  $\typeset@glyph{\slot@name} * \typeset@glyph{#1} \rightarrow
  \typeset@glyph{\slot@name} * \typeset@glyph{#2} * \typeset@glyph{#1}$}
\x@cs\def{doc-/LIG>}#1#2{%
  $\typeset@glyph{\slot@name} * \typeset@glyph{#1} \rightarrow
  \typeset@glyph{\slot@name} + \typeset@glyph{#2}$}
\x@cs\def{doc-LIG/>}#1#2{%
```

[5] With the exception of `\usedas`, which is never used at all AFAIK. It was intended to be used for math font encodings, but that is still on the `TODO` list. /LH

```
  $\typeset@glyph{\slot@name} * \typeset@glyph{#1} \rightarrow
  \typeset@glyph{#2} + \typeset@glyph{#1}$}
\x@cs\def{doc-/LIG/>}#1#2{%
  $\typeset@glyph{\slot@name} * \typeset@glyph{#1} \rightarrow
  \typeset@glyph{\slot@name} + \typeset@glyph{#2} * \typeset@glyph{#1}$}
\x@cs\def{doc-/LIG/>>}#1#2{%
  $\typeset@glyph{\slot@name} * \typeset@glyph{#1} \rightarrow
  \typeset@glyph{\slot@name} + \typeset@glyph{#2} + \typeset@glyph{#1}$}
⟨/doc⟩
```

\nextlarger
\usedas
\makerightboundary

```
⟨*pkg⟩
\let\nextlarger=\gobble_one
\let\usedas=\gobble_two
\let\makerightboundary=\gobble_one
⟨/pkg⟩
⟨*doc⟩
\def\nextlarger#1{\Bheading{Next larger} \typeset@glyph{#1}}
\def\makerightboundary#1{%
   \Bheading{Right boundary marker slot} designation \typeset@glyph{#1}%
}
⟨/doc⟩
```

Not in doc: \usedas

\vartop
\varmid
\varbot
\varrep

The ⟨*varchar commands*⟩ are:

\vartop{⟨*glyph*⟩}
\varmid{⟨*glyph*⟩}
\varbot{⟨*glyph*⟩}
\varrep{⟨*glyph*⟩}

These too do by default, which is overridden later, nothing in fontinst but typeset descriptive headings in fontdoc.

\varchar
\vartop
\varmid
\varbot
\varrep
\endvarchar

```
⟨*pkg⟩
\let\varchar=\empty_command
\let\vartop=\gobble_one
\let\varmid=\gobble_one
\let\varbot=\gobble_one
\let\varrep=\gobble_one
\let\endvarchar=\empty_command
⟨/pkg⟩
⟨*doc⟩
\def\varchar{\Bheading{Extensible glyph:}}
\def\vartop#1{\Bheading{Top} \typeset@glyph{#1}}
\def\varmid#1{\Bheading{Middle} \typeset@glyph{#1}}
\def\varbot#1{\Bheading{Bottom} \typeset@glyph{#1}}
\def\varrep#1{\Bheading{Repeated} \typeset@glyph{#1}}
\let\endvarchar\relax
⟨/doc⟩
```

\useexamplefont
\slotexample
\setslotcomment
\resetslotcomment
\unsetslotcomment
\slot@comment
\slot@font

As of v 1.8 of fontinst, we have added an interface for automatic documentation of encoding files, which has been developed by Matthias Clasen as part of his work

on math fonts. The implementation was slightly modified and integrated into this version by Ulrik Vieth.

`\setslotcomment` defines a default slot comment, stored in the variable `\slot@comment`, which is subsequently used to annotate all `\setslot` commands. The slot comment can be changed by `\resetslotcomment` or turned off by `\unsetslotcomment`.

`\useexamplefont` defines a default font, `\slot@font`, which may be referenced by calling `\slotexample` in slot comments to display the character or symbol allocated to the current slot.

Taking advantage of this mechanism, it is possible to write:

```
\useexamplefont{FONT}

\setslotcomment{The symbol `\slotexample'.}
\setslot{FOO}\endsetslot
\setslot{BAR}\endsetslot
\resetslotcomment{The character `\slotexample'.}
\setslot{BAZ}\endsetslot
```

instead of having to write:

```
\usepackage{PACKAGE-to-use-symbols-from-FONT}

\setslot{FOO}\comment{The symbol `\foo'.}\endsetslot
\setslot{BAR}\comment{The symbol `\bar'.}\endsetslot
\setslot{BAZ}\comment{The character `\baz'.}\endsetslot
```

These macros never do anything in fontinst, they just gobble their arguments.

```
⟨*pkg⟩
\let\useexamplefont=\gobble_one
\let\slotexample=\empty_command
\let\setslotcomment=\gobble_one
\let\resetslotcomment=\gobble_one
\let\unsetslotcomment=\empty_command
⟨/pkg⟩

⟨*doc⟩
\newif\ifslot@comment@
\slot@comment@false
\def\slot@comment{}
\def\setslotcomment#1{%
   \slot@comment@true
   \gdef\slot@comment{#1}}
\def\resetslotcomment#1{%
   \gdef\slot@comment{#1}}
\def\unsetslotcomment{%
   \slot@comment@false
   \gdef\slot@comment{}}

\let\slot@font=\nullfont
\def\useexamplefont#1{\font\slot@font=#1 }
\def\slotexample{{\slot@font\char\the\slot@number}}
⟨/doc⟩
```

11 Metric files

\inputmtx The macro

\inputmtx{⟨*filename*⟩}

inputs ⟨*filename*⟩.mtx, ignoring anything between \relax and \metrics, and anything after \endmetrics.

\inputmtx \metrics \endmetrics \endmetrics_text The \endmetrics_text macro expands to \endmetrics (eleven 'other' tokens, not one control sequence token). It is used instead of \string\endmetrics, which does not work since \endmetrics is outer.

```
⟨*pkg⟩
\def\inputmtx#1{
   \def\relax{\let\relax=\x_relax\iffalse}
   \let\metrics=\fi
   \primitiveinput #1.mtx\x_relax
   \let\relax=\x_relax
}
\let\metrics=\x@relax
\edef\endmetrics_text{\string\endmetrics}
\outer\def\endmetrics{\endinput}
⟨/pkg⟩
```

\inputmtx used to set \a_count, but we haven't been able to find a reason for this, so we removed it.

In fontdoc, the \metrics and \endmetrics macros initially just define a group. \inputmtx redefines them to work like in fontinst however.

```
⟨*doc⟩
\def\inputmtx#1{%
   \begingroup
      \edef\lowercase@file{\lowercase{%
         \edef\noexpand\lowercase@file{#1}%
      }}%
      \lowercase@file
      \def\relax{\let\relax=\x@relax\iffalse}
      \let\metrics=\fi
      \outer \expandafter\def \csname endmetrics\endcsname{\endinput}%
      \expandafter\input \lowercase@file.etx\x@relax
      \let\relax=\x@relax
   \endgroup
}
\def\metrics{\begingroup}
\outer\def\endmetrics{\endgroup}
⟨/doc⟩
```

\ProvidesMtxPackage The call

\ProvidesMtxPackage{⟨*package name*⟩}

signals to the package managing system that there is no need to load this package again.

```
⟨pkg⟩\def\ProvidesMtxPackage#1{\x_cs\let{pack-#1}P}
⟨*doc⟩
```

```
\def\ProvidesMtxPackage#1{%
   \Aheading{Metric package} \textsf{#1} provided.%
}
⟨/doc⟩
```

\usemtxpackage The call

\usemtxpackage{⟨*package list*⟩}

inputs those of the packages in the list that have not been loaded yet (i.e., those for which no \ProvidesMtxPackage has been made). Each package is assumed to reside in the metric file that \inputmtx loads when given the name of the package as argument.

The call

\usemtxpackage[⟨*filename*⟩]{⟨*package list*⟩}

inputs the packages in the list if at least one of them has not been loaded yet. In this case, all the packages are assumed to reside in the single metric file that \inputmtx loads when given ⟨*filename*⟩ as argument.

```
⟨*pkg⟩
\def\usemtxpackage{\futurelet\next_token\test_UseMtxPkg_arguments}
\def\test_UseMtxPkg_arguments{\ifx\next_token[
      \expandafter\mtx_package_given_file
   \else
      \expandafter\mtx_package_separate_files
   \fi
}
\newif\if_load_mtx_package_
\def\mtx_package_given_file[#1]#2{
   \_load_mtx_package_false
   \process_csep_list\load_true_unless_loaded #2,\process_csep_list,
   \if_load_mtx_package_ \inputmtx{#1} \fi
}
\def\load_true_unless_loaded#1{
   \x_cs\ifx{pack-#1}P\else\_load_mtx_package_true\fi
}
\def\mtx_package_separate_files#1{
   \process_csep_list\load_file_unless_loaded #1,\process_csep_list,
}
\def\load_file_unless_loaded#1{
   \x_cs\ifx{pack-#1}P\else \inputmtx{#1} \fi
}
⟨/pkg⟩
⟨*doc⟩
\newcommand\usemtxpackage[2][\x@relax]{%
   \ifx \x@relax#1%
      \Aheading{Require} all metric packages in
      \print@csep@list{\fam0}{\{#2\}} that has not been previously
      loaded. Expect to find them in the metric files whose names can
      be formed from the names of the packages.
   \else
      \Aheading{Require} metric packages from the file \texttt{#1.mtx}
```

```
      if some of the packages in \print@csep@list{\fam0}{\{#2\}} has not
      been loaded previously.
   \fi
}
⟨/doc⟩
```

11.1 Kerning information

\l-NAME
\r-NAME
\k
\AMOUNT

The kerning information is kept in the macros `\l-`⟨*name*⟩ and `\r-`⟨*name*⟩, containing information about how ⟨*name*⟩ kerns on the left and on the right, respectively. The `\l-`⟨*name*⟩ macro should expand out to a series of

`\k \r-`⟨*name*⟩ `\`⟨*amount*⟩

control sequences, and vice versa. Examples of `\`⟨*amount*⟩ control sequences are: `\0`, `\1`, `\1000`, `\-50`, `\33`; these corresponds to the kern amounts 0, 1, 1000 (which would be grotesquely large), −50, and 33 respectively.

\setkern

`\setkern{`⟨*glyph1*⟩`}{`⟨*glyph2*⟩`}{`⟨*integer expression*⟩`}`

Sets a kern pair between ⟨*glyph1*⟩ and ⟨*glyph2*⟩ to the specified value, which is typically a value returned by `\kerning{`⟨*glyph3*⟩`}{`⟨*glyph4*⟩`}`. If there already is a kern set between ⟨*glyph1*⟩ and ⟨*glyph2*⟩ then this will not affect the output, but it will use up another 3 units of token memory.

```
⟨*pkg⟩
\def\setkern#1#2#3{
   \x_cs\ifx{i-rawscale}\x_relax
      \expandafter\set_kern
         \csname~r-#1\expandafter\endcsname
         \csname~l-#2\endcsname{#3}
   \else
      \expandafter\set_kern
         \csname~r-#1\expandafter\endcsname
         \csname~l-#2\endcsname{\scale{#3}{\int{rawscale}}}
   \fi
}
\def\set_kern#1#2#3{
   \eval_expr{#3}
   \expandafter\set_kern_cs\csname\the\result\endcsname#1#2
}
\def\set_kern_cs#1#2#3{
   \add_to#2{\k#3#1}
   \add_to#3{\k#2#1}
}
⟨/pkg⟩
⟨*doc⟩
\def\setkern#1#2#3{%
   \Bheading{Kern} ${\typeset@glyph{#1} + \typeset@glyph{#2}}
   \rightarrow \expression0{#3}$}
⟨/doc⟩
```

\resetkern

`\resetkern{`⟨*glyph1*⟩`}{`⟨*glyph2*⟩`}{`⟨*integer expression*⟩`}`

Resets the kern pair between ⟨*glyph1*⟩ and ⟨*glyph2*⟩ to the specified value. Note however that this does not relieve TEX of the burden to remember the previous kerning pair between these two glyphs (if there was one).

```
⟨*pkg⟩
\def\resetkern#1#2#3{
   \x_cs\ifx{i-rawscale}\x_relax
      \expandafter\reset_kern
         \csname~r-#1\expandafter\endcsname
         \csname~l-#2\endcsname{#3}
   \else
      \expandafter\reset_kern
         \csname~r-#1\expandafter\endcsname
         \csname~l-#2\endcsname{\scale{#3}{\int{rawscale}}}
   \fi
}
\def\reset_kern#1#2#3{
   \eval_expr{#3}
   \expandafter\reset_kern_cs\csname\the\result\endcsname#1#2
}
\def\reset_kern_cs#1#2#3{
   \prep_to#2{\k\expandafter#3\expandafter#1}
   \prep_to#3{\k\expandafter#2\expandafter#1}
}
⟨/pkg⟩
⟨*doc⟩
\def\resetkern#1#2#3{%
   \Bheading{Override kern} ${\typeset@glyph{#1} + \typeset@glyph{#2}}
      \rightarrow \expression0{#3}$}
⟨/doc⟩
```

`\setleftkerning` `\setleftkerning{`⟨*glyph1*⟩`}{`⟨*glyph2*⟩`}{`⟨*scaled*⟩`}`
`\setrightkerning` `\setrightkerning{`⟨*glyph1*⟩`}{`⟨*glyph2*⟩`}{`⟨*scaled*⟩`}`
`\setleftrightkerning` `\setleftrightkerning{`⟨*glyph1*⟩`}{`⟨*glyph2*⟩`}{`⟨*scaled*⟩`}`

Sets left or right kerning of ⟨*glyph1*⟩ to that of ⟨*glyph2*⟩ scaled by ⟨*scaled*⟩ (which is an integer expression). `\setleftrightkerning` does both.

```
⟨*pkg⟩
\def\setleftkerning#1#2#3{
   \eval_expr_to\b_count{#3}
   \expandafter\set_kerning
      \csname~l-#1\expandafter\endcsname
      \csname~l-#2\endcsname
}
\def\setrightkerning#1#2#3{
   \eval_expr_to\b_count{#3}
   \expandafter\set_kerning
      \csname~r-#1\expandafter\endcsname
      \csname~r-#2\endcsname
}
\def\setleftrightkerning#1#2#3{
   \eval_expr_to\b_count{#3}
   \expandafter\set_kerning
      \csname~l-#1\expandafter\endcsname
      \csname~l-#2\endcsname
```

```
   \expandafter\set_kerning
      \csname~r-#1\expandafter\endcsname
      \csname~r-#2\endcsname
}
\def\set_kerning#1#2{
   \if\b_count=\one_thousand
      \def\k##1##2{
         \set_kern_cs##2##1#1
      }
   \else
      \def\k##1##2{
         \set_kern##1#1{
            \scale\b_count{\expandafter\gobble_one\string##2}
         }
      }
   \fi
   #2
}
```
⟨/pkg⟩

```
⟨*doc⟩
\def\setleftkerning#1#2#3{%
   \Bheading{Kern} \typeset@glyph{#1} on the left like
   \typeset@glyph{#2} scaled $\expression0{#3}$}
\def\setrightkerning#1#2#3{%
   \Bheading{Kern} \typeset@glyph{#1} on the right like
   \typeset@glyph{#2} scaled $\expression0{#3}$}
\def\setleftrightkerning#1#2#3{%
   \Bheading{Kern} \typeset@glyph{#1} on the left and right like
   \typeset@glyph{#2} scaled $\expression0{#3}$}
⟨/doc⟩
```

`\kerning` `\kerning{`⟨*glyph1*⟩`}{`⟨*glyph2*⟩`}`

Returns the value of kern pair between ⟨*glyph1*⟩ and ⟨*glyph2*⟩ as an integer. Returns a value of zero if such a kern pair doesn't exist.

```
⟨*pkg⟩
\def\kerning#1#2{0\x_relax
   \def\k##1{\csname~set-\string##1\endcsname\gobble_one}
   \bgroup
   \x_cs\def{set-\string\l-#2}##1##2{
      \global\result=\expandafter\gobble_one\string##2\egroup
   }
   \csname~r-#1\endcsname
   \csname~set-\string\l-#2\endcsname\gobble_one{00}
}
⟨/pkg⟩

⟨doc⟩\def\kerning#1#2{k(\typeset@glyph{#1})(\typeset@glyph{#2})}
```

`\ifiskern` `\ifiskern{`⟨*glyph1*⟩`}{`⟨*glyph2*⟩`}\then`

This command tests if there is a kern pair between ⟨*glyph1*⟩ and ⟨*glyph2*⟩. It's hard to say if there is a use for it, but it is included for symmetry.

```
⟨*pkg⟩
\def\ifiskern#1#2\then{
```

```
    \def\k##1##2{\ifx T##1 \let\k\gobble_two \fi}
    \bgroup
       \x_cs\let{l-#2}T
       \csname r-#1\endcsname
    \expandafter\egroup \ifx\k\gobble_two
}
```

⟨/pkg⟩

⟨*doc⟩

```
\def\ifiskern#1#2\then{%
    \generic@if{kern \typeset@glyph{#1}${}+{}$\typeset@glyph{#2} set}}
```

⟨/doc⟩

`\unsetkerns` `\unsetkerns{`⟨*left glyph list*⟩`}{`⟨*right glyph list*⟩`}`

This command unsets all kerns with a glyph in the ⟨*left glyph list*⟩ on the left and a glyph in the ⟨*right glyph list*⟩ on the right. The lists themselves are ordinary comma-separated lists. Unlike `\setkern` and `\resetkern`, `\unsetkerns` actually removes the kerning pairs from TEX's memory.

The implementation itself simply goes through `\r-`⟨*left glyph*⟩ for each element ⟨*left glyph*⟩ in ⟨*left glyph list*⟩ and `\l-`⟨*right glyph*⟩ for each element ⟨*right glyph*⟩ in ⟨*right glyph list*⟩, removing each `\k`⟨*token*⟩⟨*token*⟩ tripple that refers to a glyph from the opposite list as it goes along. To make this test reasonably fast, the routine first "marks" the glyphs in the other list by setting the control sequences `\slots-`⟨*glyph*⟩ to `U` (the letter token U). This mark is later removed when the `\r-`⟨*glyph*⟩ or `\l-`⟨*glyph*⟩ respectively control sequences have been gone through; the `\slots-`⟨*glyph*⟩ control sequences are then set to `\relax`.

⟨*pkg⟩

```
\def\unsetkerns#1#2{
    \let\k\k_unless_to_U
    \process_csep_list\make_slots_U#1,\process_csep_list,
    \def\do##1{\x_cs\main_remove_Us{l-##1}}
    \process_csep_list\do#2,\process_csep_list,
    \process_csep_list\make_slots_relax#1,\process_csep_list,
    \process_csep_list\make_slots_U#2,\process_csep_list,
    \def\do##1{\x_cs\main_remove_Us{r-##1}}
    \process_csep_list\do#1,\process_csep_list,
    \process_csep_list\make_slots_relax#2,\process_csep_list,
}
\def\make_slots_U#1{\x_cs\let{slots-#1}U}
\def\make_slots_relax#1{\x_cs\let{slots-#1}\x_relax}
\def\k_unless_to_U#1#2{
    \x_cs\ifx{slots-\expandafter\gobble_three\string#1}U \else
       \noexpand\k\noexpand#1\noexpand#2
    \fi
}
\def\main_remove_Us#1{
    \ifx#1\x_relax \else
       \edef#1{#1}
       \ifx#1\empty_command \let#1\x_relax \fi
    \fi
}
```

⟨/pkg⟩

⟨*doc⟩

```
\def\unsetkerns#1#2{%
   \Bheading{Remove} all kerning pairs in
   \print@csep@list{\fam0}{\{#1\}\times\{#2\}}.%
}
```
⟨/doc⟩

`\noleftkerning` `\norightkerning` `\noleftrightkerning`

The argument of these commands is the name of a glyph. The commands removes all kerns on the left, on the right, and on both sides respectively, of this glyph.

```
⟨*pkg⟩
\def\noleftkerning#1{\no_kerning{l}{#1}}
\def\norightkerning#1{\no_kerning{r}{#1}}
\def\noleftrightkerning#1{\no_kerning{l}{#1}\no_kerning{r}{#1}}
⟨/pkg⟩
⟨*doc⟩
\def\noleftkerning#1{%
   \Bheading{Remove kerns} on the left of glyphs in
   \print@csep@list{\fam0}{\{#1\}}.}
\def\norightkerning#1{%
   \Bheading{Remove kerns} on the right of glyphs in
   \print@csep@list{\fam0}{\{#1\}}.}
\def\noleftrightkerning#1{%
   \Bheading{Remove kerns} on any side of glyphs in
   \print@csep@list{\fam0}{\{#1\}}.}
⟨/doc⟩
```

`\no_kerning` This macro is called as

`\no_kerning{`⟨*side*⟩`}{`⟨*glyph list*⟩`}`

where ⟨*side*⟩ is `l` or `r`, and ⟨*glyph list*⟩ is a comma-separated list of glyph names. The macro unsets all kerns on the ⟨*side*⟩ side of the glyphs in the ⟨*glyph list*⟩.

```
⟨*pkg⟩
\def\no_kerning#1#2{
   \let\k\no_kerning_i
   \def\do##1{\csname #1-##1\endcsname}
   \bgroup
      \aftergroup\def \aftergroup\a_macro \aftergroup{
      \process_csep_list\do #2,\process_csep_list,
      \aftergroup}
   \egroup
   \def\do##1{\expandafter\let \csname #1-##1\endcsname \x_relax}
   \process_csep_list\do #2,\process_csep_list,
   \let\k\no_kerning_ii
   \def\do##1{\edef##1{##1}}
   \a_macro
}
```

`\no_kerning_i` The `\no_kerning_i` macro is used by `\no_kerning` in constructing a list of all glyphs that a glyph in the ⟨*glyph list*⟩ has a ⟨*side*⟩ kern to, while avoiding repetitions.

```
\def\no_kerning_i#1#2{
   \ifx #1\x_relax \else
      \aftergroup\do \aftergroup#1
```

```
      \let #1\x_relax
   \fi
}
```

`\no_kerning_ii` The `\no_kerning_ii` macro is similar to the `\k_unless_to_U` macro.

```
\def\no_kerning_ii#1#2{
   \ifx #1\x_relax \else \noexpand\k \noexpand#1 \noexpand#2 \fi
}
⟨/pkg⟩
```

11.2 Glyph information

`\g-NAME` The glyph information is kept in the macros `\g-`⟨*name*⟩, which expands out to:

`{`⟨*width*⟩`}{`⟨*height*⟩`}{`⟨*depth*⟩`}{`⟨*italic*⟩`}{`⟨*mapcommands*⟩`}{`⟨*mapfonts*⟩`}`

where the ⟨*mapcommands*⟩ will write out VPL `MAP` fragments to a `.vpl` file, and the ⟨*mapfonts*⟩ will produce any `MAPFONT` instructions that are needed.

`\typeset@glyph` The `\typeset@glyph` macro is analogous to `\typeset@string` and friends.

```
⟨doc⟩\let\typeset@glyph=\typeset@string
```

`\width` `\width{`⟨*glyph*⟩`}`
`\height` `\height{`⟨*glyph*⟩`}`
`\depth` `\depth{`⟨*glyph*⟩`}`
`\italic` `\italic{`⟨*glyph*⟩`}`

In fontinst, these macros return the width, height, depth, and italic correction of ⟨*glyph*⟩ as an integer.

```
⟨*pkg⟩
\def\width{\glyph_parameter\first_of_six}
\def\height{\glyph_parameter\second_of_six}
\def\depth{\glyph_parameter\third_of_six}
\def\italic{\glyph_parameter\fourth_of_six}
⟨/pkg⟩
```

In fontdoc, they print symbolic representations for the corresponding integer expressions.

```
⟨*doc⟩
\def\width#1{w(\typeset@glyph{#1})}
\def\height#1{h(\typeset@glyph{#1})}
\def\depth#1{d(\typeset@glyph{#1})}
\def\italic#1{i(\typeset@glyph{#1})}
⟨/doc⟩
```

`\mapcommands` `\mapfonts` These are similar to `\width` and its companions, but they are not user level commands.

```
⟨*pkg⟩
\def\mapcommands{\glyph_parameter\fifth_of_six}
\def\mapfonts{\glyph_parameter\sixth_of_six}
⟨/pkg⟩
```

`\glyph_parameter` `\first_of_six` `\second_of_six` `\third_of_six` `\fourth_of_six` `\fifth_of_six` `\sixth_of_six`

```
⟨*pkg⟩
```

```
\def\glyph_parameter#1#2{
   \expandafter\expandafter\expandafter
         #1\csname~g-#2\endcsname
}
\def\first_of_six#1#2#3#4#5#6{#1}
\def\second_of_six#1#2#3#4#5#6{#2}
\def\third_of_six#1#2#3#4#5#6{#3}
\def\fourth_of_six#1#2#3#4#5#6{#4}
\def\fifth_of_six#1#2#3#4#5#6{#5}
\def\sixth_of_six#1#2#3#4#5#6{#6}
```

\saved_scale
\saved_mapfont
\saved_raw
\saved_rule
\saved_special
\saved_warning
\saved_movert
\saved_moveup
\saved_push
\saved_pop

These are the commands allowed inside a glyph. They are initally set to `\relax`, so we can `\edef` with them safely.

```
\let\saved_scale\x_relax
\let\saved_mapfont\x_relax
\let\saved_raw\x_relax
\let\saved_rule\x_relax
\let\saved_special\x_relax
\let\saved_warning\x_relax
\let\saved_movert\x_relax
\let\saved_moveup\x_relax
\let\saved_push\x_relax
\let\saved_pop\x_relax
```

When the glyph is being constructed by

`\setglyph{`⟨*glyph*⟩`}` ⟨*glyph commands*⟩ `\endsetglyph`

the values of each of these parameters are kept in the following variables. Except for `\glyph_width`, these are kept globally, so they survive through `\push` ... `\pop` pairs. In addition, the current vertical offset is kept locally in `\glyph_voffset`. The `\glyph_maxhpos` variable globally keeps track of the largest horizontal offset position reached so far (with the possible exception of the current position, i.e. width, which may well be larger).

\glyph_width
\glyph_height
\glyph_depth
\glyph_italic
\glyph_map_commands
\glyph_map_fonts
\glyph_voffset
\glyph_maxhpos

```
\newcount\glyph_width
\newcount\glyph_height
\newcount\glyph_depth
\newcount\glyph_italic
\newtoks\glyph_map_commands
\newtoks\glyph_map_fonts
\newcount\glyph_voffset
\newcount\glyph_maxhpos
```

\setglyph
\gobble_glyph

The `\setglyph{`⟨*name*⟩`}` command defines `\g-`⟨*name*⟩, unless `\g-`⟨*name*⟩ has already been defined.

The reason `\g-`⟨*name*⟩ is defined before calling `\resetglyph` is to make it possible to refer to "the glyph constructed so far" using `\width{`⟨*name*⟩`}`, `\height{`⟨*name*⟩`}`, etc.

```
\def\setglyph#1{
   \ifisglyph{#1}\then
      \expandafter\gobble_glyph
```

```
   \else
      \x_cs\def{g-#1}{{\the\glyph_width}{\the
         \glyph_height}{\the\glyph_depth}{\the\glyph_italic}{\the
         \glyph_map_commands}{\the\glyph_map_fonts}}
      \resetglyph{#1}
   \fi
}
\long\def\gobble_glyph#1\endsetglyph{}
⟨/pkg⟩
⟨doc⟩\def\setglyph#1{\Aheading{Glyph `#1'}}
```

`\resetglyph` The `\resetglyph{`⟨*name*⟩`}` command redefines `\g-`⟨*name*⟩, regardless of whether `\g-`⟨*name*⟩, has already been defined.

```
⟨*pkg⟩
\def\resetglyph#1{
   \def\glyphname{#1}
   \glyph_width=0
   \global\glyph_height=0
   \global\glyph_depth=0
   \global\glyph_italic=0
   \glyph_voffset=0
   \global\glyph_maxhpos=0
   \global\glyph_map_commands={}
   \global\glyph_map_fonts={}
}
⟨/pkg⟩
⟨doc⟩\def\resetglyph#1{\Aheading{Reset glyph `#1'}}
```

`\endsetglyph`
`\endresetglyph`

```
⟨*pkg⟩
\def\endsetglyph{
   \x_cs\edef{g-\glyphname}
      {{\the\glyph_width}{\the\glyph_height}
       {\the\glyph_depth}{\the\glyph_italic}
       {\the\glyph_map_commands}{\the\glyph_map_fonts}}
}
\let\endresetglyph=\endsetglyph
⟨/pkg⟩
⟨*doc⟩
\def\endsetglyph{}
\def\endresetglyph{}
⟨/doc⟩
```

`\setrawglyph` `\setrawglyph{`⟨*glyph*⟩`}{`⟨*font*⟩`}{`⟨*size*⟩`}{`⟨*slot*⟩`}{`⟨*width*⟩`}{`⟨*height*⟩`}{`⟨*depth*⟩`}{`⟨*italic*⟩`}`

These commands are generated automatically, when an `.mtx` file is written out by `\afmtomtx`, `\pltomtx`, or `\mtxtomtx`.

```
⟨*pkg⟩
\def\setrawglyph#1#2#3#4#5#6#7#8{
   \x_cs\ifx{g-#1}\x_relax
      \x_cs\ifx{i-rawscale}\x_relax
         \x_cs\def{g-#1}{
            {#5}
```

```
            {#6}
            {#7}
            {#8}
            {\saved_raw{#2}{#4}{#1}}
            {\saved_mapfont{#2}{#3}}
         }
      \else
         \eval_expr_to\e_count{\int{rawscale}}
         \eval_expr_to\a_count{\scale{#5}\e_count}
         \eval_expr_to\b_count{\scale{#6}\e_count}
         \eval_expr_to\c_count{\scale{#7}\e_count}
         \eval_expr_to\d_count{\scale{#8}\e_count}
         \x_cs\edef{g-#1}{
            {\the\a_count}
            {\the\b_count}
            {\the\c_count}
            {\the\d_count}
            {\saved_scale{\the\e_count}
               {\saved_raw{#2}{#4}{#1}}}
            {\saved_scale{\the\e_count}
               {\saved_mapfont{#2}{#3}}}
         }
      \fi
   \fi
}
⟨/pkg⟩
⟨*doc⟩
\def\setrawglyph#1#2#3#4#5#6#7#8{\Aheading{Glyph `#1'}
   \Bheading{Taken from} slot #4 in font #2}
⟨/doc⟩
```

`\setnotglyph`

```
⟨*pkg⟩
\def\setnotglyph#1#2#3#4#5#6#7#8{
   \x_cs\ifx{g-#1-not}\x_relax
      \x_cs\ifx{i-rawscale}\x_relax
         \x_cs\def{g-#1-not}{
            {#5}
            {#6}
            {#7}
            {#8}
            {}
            {}
         }
      \else
         \eval_expr_to\e_count{\int{rawscale}}
         \eval_expr_to\a_count{\scale{#5}\e_count}
         \eval_expr_to\b_count{\scale{#6}\e_count}
         \eval_expr_to\c_count{\scale{#7}\e_count}
         \eval_expr_to\d_count{\scale{#8}\e_count}
         \x_cs\edef{g-#1-not}{
            {\the\a_count}
            {\the\b_count}
            {\the\c_count}
```

```
            {\the\d_count}
            {}
            {}
         }
      \fi
   \fi
}
⟨/pkg⟩
⟨doc⟩\def\setnotglyph#1#2#3#4#5#6#7#8{\Aheading{Pseudoglyph ‘#1-not’}}
```

`\unsetglyph` The `\unsetglyph{`⟨*name*⟩`}` command makes `\g-`⟨*name*⟩ undefined.

Not in doc: `\unsetglyph`

```
⟨pkg⟩\def\unsetglyph#1{\x_cs\let{g-#1}\x_relax}
```

11.3 Glyph commands

The ⟨*glyph commands*⟩ are:

`\glyph` `\glyph{`⟨*glyph*⟩`}{`⟨*scale*⟩`}`

```
⟨*pkg⟩
\def\glyph#1#2{
   \eval_expr_to\b_count{#2}
   \ifnum \b_count=\one_thousand
      \eval_expr_to\glyph_width{\add\glyph_width{\width{#1}}}
      \g_eval_expr_to\glyph_height{\max\glyph_height
         {\add{\height{#1}}\glyph_voffset}}
      \g_eval_expr_to\glyph_depth{\max\glyph_depth
         {\sub{\depth{#1}}\glyph_voffset}}
      \g_eval_expr_to\glyph_italic{\italic{#1}}
      \edef\a_macro{\mapcommands{#1}}
      \global\glyph_map_commands\expandafter{
         \the\expandafter\glyph_map_commands \a_macro
      }
      \edef\a_macro{\mapfonts{#1}}
   \else
      \eval_expr_to\glyph_width{
         \add\glyph_width{\scale\b_count{\width{#1}}}
      }
      \g_eval_expr_to\glyph_height{\max\glyph_height
         {\add{\scale\b_count{\height{#1}}}\glyph_voffset}}
      \g_eval_expr_to\glyph_depth{\max\glyph_depth
         {\sub{\scale\b_count{\depth{#1}}}\glyph_voffset}}
      \g_eval_expr_to\glyph_italic{\scale\b_count{\italic{#1}}}
      \edef\a_macro{\saved_scale{\the\b_count}{\mapcommands{#1}}}
      \global\glyph_map_commands\expandafter{
         \the\expandafter\glyph_map_commands \a_macro
      }
      \edef\a_macro{\saved_scale{\the\b_count}{\mapfonts{#1}}}
   \fi
   \global\glyph_map_fonts\expandafter{
      \the\expandafter\glyph_map_fonts \a_macro
   }
}
⟨/pkg⟩
```

```
⟨doc⟩\def\glyph#1#2{\Bheading{Glyph} `#1' scaled $\expression0{#2}$}
```

`\glyphrule` `\glyphrule{`⟨*width*⟩`}{`⟨*height*⟩`}`

```
⟨*pkg⟩
\def\glyphrule#1#2{
   \eval_expr_to\b_count{#1} \eval_expr_to\c_count{#2}
   \advance\glyph_width by \b_count
   \g_eval_expr_to\glyph_depth{\max\glyph_depth{-\glyph_voffset}}
   \g_eval_expr_to\glyph_height{
      \max\glyph_height{\add\glyph_voffset\c_count}
   }
   \global\glyph_italic=0
   \edef\a_macro{\saved_rule{\the\b_count}{\the\c_count}}
   \global\glyph_map_commands\expandafter{
      \the\expandafter\glyph_map_commands \a_macro
   }
}
⟨/pkg⟩
```

```
⟨*doc⟩
\def\glyphrule#1#2{%
  \Bheading{Rule} $\expression0{#1}$ by $\expression0{#2}$}
⟨/doc⟩
```

`\glyphspecial` `\glyphspecial{`⟨*string*⟩`}`
`\glyphwarning` `\glyphwarning{`⟨*string*⟩`}`

```
⟨*pkg⟩
\def\glyphspecial#1{
   \edef\a_macro{\saved_special{#1}}
   \global\glyph_map_commands\expandafter{
      \the\expandafter\glyph_map_commands \a_macro
   }
}
\def\glyphwarning#1{
   \edef\a_macro{\saved_warning{#1}}
   \global\glyph_map_commands\expandafter{
      \the\expandafter\glyph_map_commands \a_macro
   }
}
⟨/pkg⟩
```

```
⟨*doc⟩
\def\glyphspecial#1{\Bheading{Special} `#1'}
\def\glyphwarning#1{\Bheading{Warning} `#1'}
⟨/doc⟩
```

`\movert` `\movert{`⟨*xoffset*⟩`}`
`\moveup` `\moveup{`⟨*yoffset*⟩`}`

```
⟨*pkg⟩
\def\movert#1{
   \eval_expr{#1}
   \ifnum \glyph_maxhpos<\glyph_width
      \global\glyph_maxhpos\glyph_width
   \fi
```

```
   \ifnum 0=\result \else
      \advance\glyph_width by \result
      \edef\a_macro{\saved_movert{\the\result}}
      \global\glyph_map_commands\expandafter{
         \the\expandafter\glyph_map_commands \a_macro
      }
   \fi
}
\def\moveup#1{
   \eval_expr{#1}
   \ifnum\result=0\else
      \advance\glyph_voffset by \result
      \edef\a_macro{\saved_moveup{\the\result}}
      \global\glyph_map_commands\expandafter{
         \the\expandafter\glyph_map_commands \a_macro
      }
   \fi
}
⟨/pkg⟩
⟨*doc⟩
\def\movert#1{\Bheading{Rt}~$\expression0{#1}$}
\def\moveup#1{\Bheading{Up}~$\expression0{#1}$}
⟨/doc⟩
```

\push
\pop

```
⟨*pkg⟩
\def\push{
   \bgroup
   \global\glyph_map_commands\expandafter{
      \the\glyph_map_commands \saved_push
   }
}
\def\pop{
   \ifnum \glyph_maxhpos<\glyph_width
      \global\glyph_maxhpos\glyph_width
   \fi
   \egroup
   \global\glyph_map_commands\expandafter{
      \the\glyph_map_commands \saved_pop
   }
}
⟨/pkg⟩
⟨*doc⟩
\def\push{\Bheading{Push}}
\def\pop{\Bheading{Pop}}
⟨/doc⟩
```

\resetwidth \resetwidth{⟨*width*⟩}
\resetheight \resetheight{⟨*height*⟩}
\resetdepth \resetdepth{⟨*depth*⟩}
\resetitalic \resetitalic{⟨*italic*⟩}

```
⟨*pkg⟩
```

```
\def\resetwidth#1{\movert{\sub{#1}\glyph_width}}
\def\resetheight{\g_eval_expr_to\glyph_height}
\def\resetdepth{\g_eval_expr_to\glyph_depth}
\def\resetitalic{\g_eval_expr_to\glyph_italic}
⟨/pkg⟩
⟨*doc⟩
\def\resetwidth#1{\Bheading{Reset width to $\expression0{#1}$}}
\def\resetheight#1{\Bheading{Reset height to $\expression0{#1}$}}
\def\resetdepth#1{\Bheading{Reset depth to $\expression0{#1}$}}
\def\resetitalic#1{\Bheading{Reset italic to $\expression0{#1}$}}
⟨/doc⟩
```

`\glyphpcc` Some syntactic sugar: `\glyphpcc` could do with optimization.
`\glyphpcc{`⟨*glyph*⟩`}{`⟨*xoffset*⟩`}{`⟨*yoffset*⟩`}`

```
⟨*pkg⟩
\def\glyphpcc#1#2#3{
   \push
      \movert{#2}
      \moveup{#3}
      \glyph{#1}{\one_thousand}
   \pop
}
⟨/pkg⟩
⟨*doc⟩
\def\glyphpcc#1#2#3{
   \push
      \movert{#2}
      \moveup{#3}
      \glyph{#1}{1000}
   \pop
}
⟨/doc⟩
```

`\glyphhboxright` The `\glyphhboxright` command is a valid integer expression, but its value is only meaningful inside a `\setglyph` ... `\endsetglyph` or `\resetglyph` ... `\endresetglyph` structure. There its value is almost the horizontal position of the right edge of the bounding box for the glyph constructed so far—in reality it is the greatest horizontal offset reached so far.

```
⟨pkg⟩\def\glyphhboxright{\max\glyph_width\glyph_maxhpos}
⟨doc⟩\def\glyphhboxright{\hbox{glyphhboxright}}
```

`\samesize` `\samesize{`⟨*glyph*⟩`}`

```
⟨*pkg⟩
\def\samesize#1{
   \x_cs\ifx{g-#1}\x_relax
      \resetwidth{\glyphhboxright}
   \else
      \expandafter\expandafter\expandafter
         \same_size\csname g-#1\endcsname
   \fi
}
\def\same_size#1#2#3#4#5#6{
```

```
    \movert{\sub{#1}\glyph_width}
    \global\glyph_height=#2
    \global\glyph_depth=#3
    \global\glyph_italic=#4
}
⟨/pkg⟩
⟨doc⟩\def\samesize#1{\Bheading{Reset dimensions to those of ‘#1’.}}
```

`\ifisglyph` The control flow command:

```
⟨*pkg⟩
\def\ifisglyph#1\then{
    \x_cs\ifx{g-#1}\x_relax
       \expandafter\if_false
    \else
       \expandafter\if_true
    \fi
}
⟨/pkg⟩
⟨doc⟩\def\ifisglyph#1\then{\generic@if{glyph \typeset@glyph{#1} set}}
```

12 Converting an ETX file to a (V)PL file

12.1 Lowest-level user interface

`\etxtovpl` The macro:

> `\etxtovpl{`⟨*encoding*⟩`}{`⟨*vplfile*⟩`}`

writes a virtual font (as a virtual property list) with the encoding ⟨*encoding*⟩. (This macro is called by `\installfont`.)

`\etxtopl` The macro:

> `\etxtopl{`⟨*encoding*⟩`}{`⟨*plfile*⟩`}`

writes a font (as a property list) with the encoding ⟨*encoding*⟩. (This macro is called by `\installrawfont`.)

`\etxtovpl`
`\etxtopl`

```
⟨*pkg⟩
\def\etxtovpl#1#2{{
    \def\vpl_extension{vpl}
    \def\vpl_title{COMMENT}
    \def\vpl_font{virtual~font}
    \def\vpl_Font{Virtual~font}
    \def\vpl_caller{\string\etxtovpl}
    \def\vpl_to_vf##1{vpltovf~##1.vpl~##1.vf~##1.tfm}
    \etx_to_font{#1}{#2}
}}
\def\etxtopl#1#2{{
    \def\vpl_extension{pl}
    \def\vpl_title{COMMENT}
    \def\vpl_font{font}
    \def\vpl_Font{Font}
```

```
    \def\vpl_caller{\string\etxtopl}
    \def\vpl_to_vf##1{pltotfm~##1.pl~##1.tfm}
    \_including_map_false
    \etx_to_font{#1}{#2}
}}
```

`\if_including_map_` LH 1998/10/02: The v 1.5 method of controlling whether `MAP` and `MAPFONT` property lists should be written requires that some code is put in an awkward place (in `\make_mapfonts` rather than `\make_characters`). As I am reorganising the code anyway, I saw it best to let a switch control this instead. If `\if_including_map_` is true, `MAP` and `MAPFONT` property lists are written to the file (which should be a VPL), if it is false then they are not written to the file (which is then a normal PL).

```
\newif\if_including_map_
\_including_map_true
```

`\etx_to_font` `\etx_to_font{`⟨*encoding*⟩`}{`⟨*fontfile*⟩`}` makes a `.vpl` file.

```
\def\etx_to_font#1#2{
    \make_assignments{#1}
    \open_out{#2.\vpl_extension}
    \out_line{(\vpl_title\space\vpl_font\space
       #2~created~by~fontinst~v\fontinstversion)}
    \out_line{}
    \out_line{(COMMENT~Filename:~#2.\vpl_extension)}
    \out_line{(COMMENT~Created~by:~tex~\jobname)}
    \out_line{(COMMENT~Created~using:~\vpl_caller{#1}{#2})}
    \out_line{}
    \out_line{(COMMENT~This~file~can~be~turned~into~a~\vpl_font\space
       with)}
    \out_line{(COMMENT~\vpl_to_vf{#2})}
    \out_line{}
    \out_line{(COMMENT~THIS~FILE~CAN~THEN~BE~DELETED.)}
    \out_line{}
    \make_header{#1}
    \if_including_map_ \make_mapfonts{#1} \fi
    \make_fontdimens{#1}
    \make_ligtable{#1}
    \make_characters{#1}
    \make_tidy{#1}
    \out_line{}
    \out_line{(COMMENT~END~OF~FILE~#2.\vpl_extension)}
    \close_out{\vpl_Font}
}
```

`\pre_first_etx_pass_hook`
`\pre_second_etx_pass_hook`
`\pre_third_etx_pass_hook`
`\pre_fourth_etx_pass_hook`
`\post_first_etx_pass_hook`
`\post_second_etx_pass_hook`
`\post_third_etx_pass_hook`
`\post_fourth_etx_pass_hook`
`\tidying_up_hook`

LH 1998/09/30: As the `\etxtovpl` and `\etxtopl` macros are the lowest-level interfaces to what they do that are available in fontinst and as the amount of code they execute is really quite large, I feel there is a need for some robust mechanism for adding code to this, in case the need should arise. Such a mechanism is provided by the following hooks:

```
\let\pre_first_etx_pass_hook\x_relax
\let\pre_second_etx_pass_hook\x_relax
\let\pre_third_etx_pass_hook\x_relax
\let\pre_fourth_etx_pass_hook\x_relax
```

```
\let\post_first_etx_pass_hook\x_relax
\let\post_second_etx_pass_hook\x_relax
\let\post_third_etx_pass_hook\x_relax
\let\post_fourth_etx_pass_hook\x_relax
\let\tidying_up_hook\x_relax
```

To include code in one of them, one should write things like

\add_to\pre_first_etx_pass_hook{⟨*extra code*⟩}

12.2 Glyph to slot assignments

The way fontinst has traditionally kept track of which glyph goes to which slot—information which is needed when writing the LIGTABLE in a (V)PL, for \nextlarger, and for the varchar commands—is by storing the slot number in the integer whose name is the name of the glyph. There are two main problems with this implementation: (i) If a user sets an integer which happens to have the same name as a glyph that is not assigned to any slot, then kerns for that unused glyph might be written to the file, accidentally creating a kerning pair where there should be none. (ii) It is only possible to store one slot number per glyph, so if one uses the same glyph for several slots then fontinst can only write kerns (and ligatures) for one of the occurences, despite the fact that all occurences have the same typographical need for them. These problems do only occur for glyphs on the right side of a ligature/kerning pair, but they are still serious enough. This part of fontinst has therefore been reimplemented.

The new implementation does not use integers, instead it uses a new family
\slots-GLYPH of macros with names on the form \slots-⟨*glyph*⟩. These macros expand to sequences of \do⟨*character*⟩, where ⟨*character*⟩ is a category 12 (other) token whose character code equals the slot number.

In the entire space of such names, each slot sould be mensioned at most once, with one exception, namely the slot which serves as right boundary marker, which may occur twice (once for the glyph which actually is assigned to the slot and once for the right boundary marker glyph). To detect whether there is a collision between these two uses of the slot, the right boundary marker glyph uses \rboundary_do instead of \do and the glyph whose slot serves as boundarychar uses \rbserver_do instead of \do. By redefining these two control sequences to generate a warning message when appropriate, the occurence of such a collision can be brought to the user's attention.

As it turned out to not be such a big deal, some extra code (protected by a docstrip switch called ⟨boundaryCompability⟩) that makes fontinst compatible with the previous syntax for boundary handling has been included. I don't recommend using it though, since it makes fontinst store almost all slot numbers in two places.

\make_assignments

```
\def\make_assignments#1{
   \let\do_slot=\assign_slot
   \let\end_do_slot=\end_assign_slot
   \def\do_boundary{\bgroup
      \let\makerightboundary=\bad_makerightboundary
   }
   \let\endsetleftboundary=\egroup
   \let\makerightboundary=\assign_rboundary
```

```
    \pre_first_etx_pass_hook
    \inputetx{#1}
    \post_first_etx_pass_hook
    \let\end_do_slot=\empty_command
    \let\do_boundary=\x_relax
    \let\endsetleftboundary=\x_relax
    \let\makerightboundary=\gobble_one
}
```

`\assign_slot` `\assign_slot` begins the assignment of a slot to a glyph. `\end_assign_slot` completes it. In the time between these two, some information is stored in `\a_toks`, so that it can be modified by an intervening `\makerightboundary` (which is then `\assign_rboundary`).

`\end_assign_slot`

Note that the code below will not reset `\lccode0`. The value of this register should be considered uncertain while a ligful PL or VPL file is written.

```
\def\assign_slot{\a_toks={\do}}
\begingroup
    \catcode0=12
    \gdef\end_assign_slot{
       \ifisglyph\slot_name\then
          \lccode0=\slot_number
          \lowercase{
             \expandafter\add_to
                \csname slots-\slot_name\expandafter\endcsname
                \expandafter{\the\a_toks^^@}
          }
⟨boundaryCompability⟩          \x_resetint\slot_name\slot_number
       \fi
    }
```

`\assign_rboundary` `\assign_rboundary` is what `\makerightboundary` is when it is assigning a slot to act as right boundary marker.

`\bad_makerightboundary`

`\bad_makerightboundary` is what `\makerightboundary` temporarily gets set to between `\setleftboundary` and `\endsetleftboundary`.

```
    \gdef\assign_rboundary#1{
       \lccode0=\slot_number
       \lowercase{\x_cs\add_to{slots-#1}{\rboundary_do^^@}}
       \x_setint{\percent_char boundarychar}\slot_number
       \a_toks={\rbserver_do}
    }
\endgroup
\def\bad_makerightboundary#1{
    \errhelp={The~left~boundary~is~not~a~slot,~so~it~cannot~serve~
       as~right~boundary.}
    \errmessage{Incorrect~use~of~\string\makerightboundary}
}
```

`\get_slot_num` It is sometimes necessary to get the number of an arbitrary slot given only the name of the glyph which has been assigned to the slot. In such situations, the call

`\get_slot_num{`⟨*glyph*⟩`}`

will set `\result` to the number of one such slot if the glyph has been assigned to some slot, or set `\result` to minus one if the glyph has not been assigned to a slot.

```
\def\get_slot_num#1{
⟨*!boundaryCompability⟩
   \global\result=-1
   \bgroup
      \def\do{\global\result=‘}
      \let\rbserver_do=\do
      \let\rboundary_do=\gobble_one
      \csname slots-#1\endcsname
   \egroup
⟨/!boundaryCompability⟩
⟨*boundaryCompability⟩
   \ifisint{#1}\then
      \global\result=\int{#1}
   \else
      \global\result=-1
   \fi
   \x_relax
⟨/boundaryCompability⟩
}
```

`\ifisinslot` The call `\ifisinslot{`⟨*glyph*⟩`}{`⟨*slot*⟩`}\then` can be used to test whether glyph ⟨*glyph*⟩ has been assigned to slot ⟨*slot*⟩. Both then-part and else-part of the conditional will however be ignored when the ETX file is read the first time, since the assignment of glyph to slots need not have been completed yet.

```
\def\ifisinslot#1#2\then{
   \ifx \makerightboundary\gobble_one
```

This is used to test if the ETX file is being read in for the first time. It is a bit hacky, but it is efficient.

```
      \eval_expr{#2}
      \begingroup
         \def\do##1{\ifnum ‘##1=\result \let\do=\gobble_one \fi}
         \def\rbserver_do{\do}
         \let\rboundary_do=\gobble_one
         \csname slots-#1\endcsname
      \expandafter\endgroup
      \ifx \do\gobble_one
         \expandafter\expandafter \expandafter\if_true
      \else
         \expandafter\expandafter \expandafter\if_false
      \fi
   \else
      \expandafter\gobble_if
   \fi
}
⟨/pkg⟩
⟨*doc⟩
\def\ifisinslot#1#2\then{%
   \generic@if{glyph \typeset@glyph{#1} assigned to slot
   $\expression0{#2}$}%
}
⟨/doc⟩
```

12.3 The header, mapfonts, and fontdimens

\scaled_design_size
\afmconvert

The call

\afmconvert⟨*dimen*⟩=⟨*integer expression*⟩;

converts a count into a dimen, assuming the count is a number of AFM units. I'll assume that the largest dimension I'll have to deal with is 131 pt, to try to minimize rounding errors.

```
⟨*pkg⟩
\newdimen\scaled_design_size
\def\afm_convert#1=#2;{
   \eval_expr{#2}
   #1=\scaled_design_size
   \divide#1 by 8
   \multiply #1 by \result
   \divide #1 by \one_thousand
   \multiply#1 by 8
}
```

\vpl_real
\vpl_int

The commands \vpl_real⟨*dimen*⟩ and \vpl_int⟨*count*⟩ print a dimension and integer respectively in (V)PL syntax.

```
\def\vpl_real#1{R~\expandafter\lose_measure\the#1}
\def\vpl_int#1{D~\the#1}
```

LH 1998/12/04: I have removed the count register \boundary_char as the reimplementation of the left and right boundaries have drastically decreased the need to check which slot serves as the right boundary. Instead the integer %boundarychar (whose name cannot normally be typed) is used to store the number of this slot, but the user need never (and should not) access this integer directly. If the integer is not set then no slot serves as boundarychar.

\side_bearings
\curr_bearings

These two dimens are used by the letterspacing mechanism.

```
\newdimen\side_bearings
\newdimen\curr_bearings
```

\make_header

```
\def\make_header#1{
   \global\font_count=0
   \setdim{designsize}{10pt}
   \scaled_design_size=\dim{designsize}
   \out_line{(DESIGNSIZE~\vpl_real\scaled_design_size)}
   \out_line{(DESIGNUNITS~\vpl_real\scaled_design_size)}
   \x_setstr{codingscheme}{UNKNOWN}
   \out_line{(CODINGSCHEME~\str{codingscheme})}
⟨*boundaryCompability⟩
   \ifisint{boundarychar}\then
      \x_setint{\percent_char boundarychar}{\int{boundarychar}}
      \immediate\write16{Please~use~\string\setleftboundary\space
         and/or~\string\makerightboundary^^J
         instead~of~setting~the~boundarychar~integer.}
   \fi
⟨/boundaryCompability⟩
```

```
    \ifisint{\percent_char boundarychar}\then
       \a_count=\int{\percent_char boundarychar}
       \out_line{(BOUNDARYCHAR~\vpl_int\a_count)}
    \fi
    \x_setint{letterspacing}{0}
    \afm_convert\side_bearings=\int{letterspacing};
    \divide \side_bearings 2
    \x_setint{minimumkern}{0}
    \minimum_kern=\int{minimumkern}
    \out_line{}
}
```

`\make_mapfonts`

```
\def\make_mapfonts#1{
    \let\saved_scale\vpl_scale
    \let\saved_mapfont\vpl_mapfont
    \let\do_slot=\do_mapfont
    \pre_second_etx_pass_hook
    \inputetx{#1}
    \post_second_etx_pass_hook
    \out_line{}
}
```

`\do_mapfont` `\do_mapfont` produces a `MAPFONT` entry for each font used by glyph `\slot_name`.

```
\def\do_mapfont{
    \ifisglyph\slot_name\then
       \mapfonts\slot_name
    \fi
}
```

The following commands can be used in the ⟨*mapfonts*⟩ glyph parameter.

`\vpl_scale`

```
\def\vpl_scale#1#2{{
    \divide \scaled_design_size by 8
    \multiply \scaled_design_size by #1
    \divide \scaled_design_size by \one_thousand
    \multiply \scaled_design_size by 8
    #2
}}
```

`\vpl_mapfont`

```
\def\vpl_mapfont#1#2{
    \a_dimen=#2
    \x_cs\ifx{#1-\the\scaled_design_size}\x_relax
       \x_cs\xdef{#1-\the\scaled_design_size}{\the\font_count}
       \x_cs\xdef{f-\the\font_count}{#1-\the\scaled_design_size}
       \out_line{(MAPFONT~\vpl_int\font_count\space
          (FONTNAME~#1)~
          (FONTDSIZE~\vpl_real\a_dimen)~
          (FONTAT~\vpl_real\scaled_design_size))}
       \record_usage{#1}
       \global\advance\font_count by 1
    \fi
}
```

```
  \font_count
 \next_mapfont 1263 \newcount\font_count
 \prev_mapfont 1264 \newcount\next_mapfont
               1265 \newcount\prev_mapfont

\make_fontdimens
               1266 \def\make_fontdimens#1{
               1267    \out_line{(FONTDIMEN}
               1268       \ifisint{fontdimen(1)}\then
               1269          \a_dimen=\int{fontdimen(1)}pt
               1270          \divide\a_dimen by \one_thousand
               1271          \out_lline{(PARAMETER~D~1~\vpl_real\a_dimen)}
               1272       \fi
               1273       \a_count=2
               1274       \loop\ifnum\a_count<256
               1275          {\ifisint{fontdimen(\the\a_count)}\then
               1276             \afm_convert\a_dimen=\int{fontdimen(\the\a_count)};
               1277             \out_lline{(PARAMETER~\vpl_int\a_count\space
               1278                \vpl_real\a_dimen)}
               1279          \fi}
               1280       \advance\a_count by 1 \repeat
               1281    \out_lline{)}
               1282    \out_line{}
               1283 }
```

12.4 The ligtable

```
  \make_ligtable
               1284 \def\make_ligtable#1{
               1285    \bgroup
               1286       \out_line{(LIGTABLE}
               1287 ⟨!boundaryCompability⟩       \let\do_slot=\bgroup
               1288 ⟨*boundaryCompability⟩
               1289       \def\do_slot{\bgroup
               1290          \ifisint{boundarychar}\then
               1291             \ifnum \int{boundarychar}=\slot_number
               1292                \def\vpl_liglabel{\out_liglabel\boundary_liglabel}
               1293             \fi
               1294          \fi
               1295       }
               1296 ⟨/boundaryCompability⟩
               1297       \let\end_do_slot=\vpl_kerning
               1298       \def\do_boundary{\bgroup \let\vpl_liglabel=\boundary_liglabel}
               1299       \let\endsetleftboundary=\vpl_kerning
               1300       \let\ligature=\vpl_ligature
               1301       \let\k=\vpl_kern
               1302       \let\rbserver_do=\vpl_rbserver_do
               1303       \let\rboundary_do=\vpl_rboundary_do
               1304       \pre_third_etx_pass_hook
               1305       \inputetx{#1}
               1306       \post_third_etx_pass_hook
               1307       \out_lline{)}
               1308    \egroup
```

```
    \out_line{}
}
```

\vpl_rbserver_do
\vpl_rboundary_do
\wrn_rboundary_do

`\vpl_rbserver_do` and `\vpl_rboundary_do` are what `\rbserver_do` and `\rboundary_do` respectively are when a ligkern program is being written and no entry has yet been written for the boundarychar slot. `\wrn_rboundary_do` is what one of them will get set to afterwards, so that warnings are written when collisions between using the slot as a right boundary marker and using the slot for the actual glyph occur.

```
\def\vpl_rbserver_do#1{
    \do{#1}
    \let\rboundary_do=\wrn_rboundary_do
}
\def\vpl_rboundary_do#1{
    \do{#1}
    \let\rbserver_do=\wrn_rboundary_do
}
\def\wrn_rboundary_do#1{
    \do{#1}
    \immediate\write16{Boundarychar~slot~usage~collision~in~
       `\slot_name'~ligkern~program.}
}
```

\vpl_ligature

`\vpl_ligature{`⟨*type*⟩`}{`⟨*glyph*⟩`}{`⟨*glyph*⟩`}` produces a ligtable entry for glyph `\slot_name`.

The double spaces in the `\out_lline` statement below might look strange, but `\number` will always gobble the first one.

```
\def\vpl_ligature#1#2#3{
    \get_slot_num{#3}
    \ifnum -1=\result
       \immediate\write16{Warning:~\string\ligature\space
          for~unknown~slot~`#3'.}
    \else
       \x_cs\ifx{slots-#2}\x_relax
          \immediate\write16{Warning:~\string\ligature\space
             for~unknown~slot~`#2'.}
       \else
          \def\do##1{
             \vpl_liglabel
             \out_lline{(#1~D~\number`##1~\space\vpl_int\result)~
                (COMMENT~#2~#3)}
          }
          \csname slots-#2\endcsname
       \fi
    \fi
}
```

\vpl_kerning

`\vpl_kerning` writes out kerning instructions.

```
\def\vpl_kerning{
    \let\do=\vpl_kern_do
    \csname~r-\slot_name\endcsname
    \vpl_ligstop
    \egroup
}
```

\vpl_kern `\vpl_kern \l-⟨name⟩ \⟨amount⟩` writes out a KRN instruction.

`\vpl_kern` has been modified so that at most one KRN instruction is written for each (ordered) pair of characters. The idea is basically to make fontinst forget, until the end of `\vpl_kerning`, that the glyph for which `\vpl_kern` is being called has been assigned a slot, as this stops any more KRN instructions for that particular glyph from being written. `\vpl_kern` has also been modified so that it will not write out any KRN instructions for kerns whose absolute value is less than or equal to `\minimum_kern`. `\minimum_kern` gets set to the value of the integer `minimumkern` in `\make_header`. If the user has not set `minimumkern`, a default value of 0 will be supplied by fontinst.

```
\def\vpl_kern#1#2{
   \edef\a_macro{\expandafter\gobble_three\string#1}
   \a_count=\expandafter\gobble_one\string#2\x_relax
   \ifnum -\a_count>\a_count
      \a_count=-\a_count
   \fi
   \ifnum \a_count>\minimum_kern
      \afm_convert\a_dimen=\expandafter\gobble_one\string#2;
      \csname slots-\a_macro\endcsname
   \fi
   \x_cs\let{slots-\a_macro}=\x_relax
}
```

Observation (LH 1999/03/16): The above construction has the side-effect that a direct contradiction between a right boundary glyph and a glyph in the %boundarychar slot won't be detected if one of the items in contradiction is a kern less than or equal to `\minimum_kern`. Perhaps this should be changed (letting `\do` equal to `\gobble_one` would let the dectection mechanism work, but `\do` would have to be restored afterwards and it is doubtful if it is worth it).

\vpl_kern_do `\vpl_kern_do` is what `\do` is defined to be when kerns are written.

```
\def\vpl_kern_do#1{
   \vpl_liglabel
   \out_lline{
      (KRN~D~\number`#1~\space\vpl_real\a_dimen)~
      (COMMENT~\a_macro)
   }
}
```

\out_liglabel `\out_liglabel` writes out a LIGLABEL instruction for a slot. `\boundary_liglabel` writes out a LIGLABEL instruction for the BOUNDARYCHAR ligkern program. `\vpl_liglabel` writes out the correct LIGLABEL instruction for the current ligkern program, if it is approriate.

```
\def\out_liglabel{
   \out_lline{(LABEL~\vpl_int\slot_number)~(COMMENT~\slot_name)}
   \let\vpl_liglabel=\x_relax
   \let\vpl_ligstop=\out_ligstop
}
\def\boundary_liglabel{
   \out_lline{(LABEL~BOUNDARYCHAR)~(COMMENT~\slot_name)}
   \let\vpl_liglabel=\x_relax
   \let\vpl_ligstop=\out_ligstop
```

```
}
\let\vpl_liglabel=\out_liglabel
```

\out_ligstop
\vpl_ligstop

\vpl_ligstop writes out a LIGSTOP instruction if appropriate.

```
\def\out_ligstop{\out_lline{(STOP)}
   \let\vpl_liglabel=\out_liglabel
   \let\vpl_ligstop=\x_relax}
\let\vpl_ligstop=\x_relax
```

12.5 The characters

\make_characters

```
\def\make_characters#1{
   \bgroup
      \let\do_slot=\do_character
      \let\end_do_slot=\end_do_character
      \let\nextlarger=\vpl_nextlarger
      \let\varchar=\vpl_varchar
      \let\endvarchar=\end_vpl_varchar
      \let\vartop=\vpl_vartop
      \let\varmid=\vpl_varmid
      \let\varbot=\vpl_varbot
      \let\varrep=\vpl_varrep
      \if_including_map_
         \let\saved_raw\vpl_raw
         \let\saved_rule\vpl_rule
         \let\saved_special\vpl_special
         \let\saved_warning\vpl_warning
         \let\saved_movert\vpl_movert
         \let\saved_moveup\vpl_moveup
         \let\saved_push\vpl_push
         \let\saved_pop\vpl_pop
      \else
         \let\do_character_map\x_relax
      \fi
      \pre_fourth_etx_pass_hook
      \inputetx{#1}
      \post_fourth_etx_pass_hook
   \egroup
}
```

\do_character

\do_character produces a character entry for glyph \slot_name in slot \slot_number.

```
\def\do_character{
   \x_cs\ifx{g-\slot_name}\x_relax
      \expandafter\gobble_setslot
   \else
      \ifx\slot_name\notdef_name\else
         \out_line{(CHARACTER~\vpl_int\slot_number\space
            (COMMENT~\slot_name)}
         \afm_convert\a_dimen=\width\slot_name;
         \do_character_sidebearings
         \out_lline{(CHARWD~\vpl_real\a_dimen)}
         \afm_convert\a_dimen=\height\slot_name;
```

```
         \out_lline{(CHARHT~\vpl_real\a_dimen)}
         \afm_convert\a_dimen=\depth\slot_name;
         \out_lline{(CHARDP~\vpl_real\a_dimen)}
         \afm_convert\a_dimen=\italic\slot_name;
         \ifnum\a_dimen>0
            \out_lline{(CHARIC~\vpl_real\a_dimen)}
         \fi
         \do_character_map
      \fi
   \fi
}
```

\do_character_sidebearings

```
\def\do_character_sidebearings{
   \ifisint{\slot_name-spacing}\then
      \afm_convert\curr_bearings=\int{\slot_name-spacing};
      \divide\curr_bearings by 2
   \else
      \curr_bearings=\side_bearings
   \fi
   \afm_convert\a_dimen=\width\slot_name;
   \advance\a_dimen by 2\curr_bearings
}
```

\do_character_map

```
\def\do_character_map{
   \global\prev_mapfont=0 \out_lline{(MAP}
      \ifdim 0pt=\curr_bearings
         \mapcommands\slot_name
      \else
         \out_llline{(MOVERIGHT~\vpl_real\curr_bearings)}
         \mapcommands\slot_name
         \out_llline{(MOVERIGHT~\vpl_real\curr_bearings)}
      \fi
   \out_llline{)}
}
```

\do_character_no_letterspacing

LH 1999/03/27: This is an alternative version of \do_character which I think is now antiquated. It used to have an advantage over \do_character in that it did not do any letterspacing (in v 1.5, that was good since \do_character used to write code for letterspacing to the VPL regardless of whether is was needed or not), but that advantage is gone now since \do_character_map is a bit less stupid now than it used to be.

For the record, I don't think there ever was a user interface for using this macro instead of \do_character, but I suspect there are plenty of people around who have hacked it into being used.

```
⟨*obsolete⟩
\def\do_character_no_letterspacing{
   \x_cs\ifx{g-\slot_name}\x_relax
      \expandafter\gobble_setslot
   \else
      \ifx\slot_name\notdef_name\else
         \out_line{(CHARACTER~\vpl_int\slot_number\space
```

```
            (COMMENT~\slot_name)}
         \afm_convert\a_dimen=\width\slot_name;
         \out_lline{(CHARWD~\vpl_real\a_dimen)}
         \afm_convert\a_dimen=\height\slot_name;
         \out_lline{(CHARHT~\vpl_real\a_dimen)}
         \afm_convert\a_dimen=\depth\slot_name;
         \out_lline{(CHARDP~\vpl_real\a_dimen)}
         \afm_convert\a_dimen=\italic\slot_name;
         \ifnum\a_dimen>0 \out_lline{(CHARIC~\vpl_real\a_dimen)} \fi
         \global\prev_mapfont=0 \out_lline{(MAP}
            \mapcommands\slot_name
         \out_llline{)}
      \fi
   \fi
}
⟨/obsolete⟩
```

\gobble_setslot

```
\long\def\gobble_setslot#1\endsetslot{\endsetslot}
```

\end_do_character

```
\def\end_do_character{
   \ifisglyph\slot_name\then
      \out_lline{)}
   \fi
}
```

\notdef_name

```
\def\notdef_name{.notdef}
```

12.6 Slot commands that put things in a character property list

Here follows the active definitions for those slot commands that causes things to be put in CHARACTER property lists.

\vpl_nextlarger `\vpl_nextlarger{`⟨*name*⟩`}` produces a NEXTLARGER entry.

```
\def\vpl_nextlarger#1{
   \get_slot_num{#1}
   \ifnum -1<\result
      \out_lline{(NEXTLARGER~D~\the\result)~(COMMENT~#1)}
   \else
      \immediate\write16{Warning:~\string\nextlarger\space
         for~unknown~slot~`#1'}
   \fi
}
```

\vpl_varchar \end_vpl_varchar `\vpl_varchar` ⟨*varchar commands*⟩ `\end_vpl_varchar` produces a VARCHAR entry.

```
\def\vpl_varchar{\out_lline{(VARCHAR}}
\def\end_vpl_varchar{\out_llline{)}}
```

\vpl_vartop

```
\def\vpl_vartop#1{
   \get_slot_num{#1}
   \ifnum -1<\result
      \out_llline{(TOP~D~\the\result)~(COMMENT~#1)}
   \else
      \immediate\write16{Warning:~\string\vartop\space
         for~unknown~slot~`#1'}
   \fi
}
```

\vpl_varmid

```
\def\vpl_varmid#1{
   \get_slot_num{#1}
   \ifnum -1<\result
      \out_llline{(MID~D~\the\result)~(COMMENT~#1)}
   \else
      \immediate\write16{Warning:~\string\varmid\space
         for~unknown~slot~`#1'}
   \fi
}
```

\vpl_varbot

```
\def\vpl_varbot#1{
   \get_slot_num{#1}
   \ifnum -1<\result
      \out_llline{(BOT~D~\the\result)~(COMMENT~#1)}
   \else
      \immediate\write16{Warning:~\string\varbot\space
         for~unknown~slot~`#1'}
   \fi
}
```

\vpl_varrep

```
\def\vpl_varrep#1{
   \get_slot_num{#1}
   \ifnum -1<\result
      \out_llline{(REP~D~\the\result)~(COMMENT~#1)}
   \else
      \immediate\write16{Warning:~\string\varrep\space
         for~unknown~slot~`#1'}
   \fi
}
```

12.7 Saved map commands

The following commands (and `\vpl_scale`, which is defined above) can be used in the ⟨*mapcommands*⟩ glyph parameter.

\vpl_raw

```
\def\vpl_raw#1#2#3{
   \global\next_mapfont=\csname#1-\the\scaled_design_size\endcsname
   \x_relax
```

```
    \ifnum\next_mapfont=\prev_mapfont\else
       \out_llline{(SELECTFONT~\vpl_int\next_mapfont)~
          (COMMENT~#1~at~\the\scaled_design_size)}
    \fi
    \out_llline{(SETCHAR~D~#2)~(COMMENT~#3)}
    \global\prev_mapfont=\next_mapfont
}
```

\vpl_rule

```
\def\vpl_rule#1#2{
    \afm_convert\a_dimen=#1;
    \afm_convert\b_dimen=#2;
    \out_llline{(SETRULE~\vpl_real\b_dimen\space~\vpl_real\a_dimen)}
}
```

\vpl_special
\vpl_warning

```
\def\vpl_special#1{
    \out_llline{(SPECIAL~#1)}}
\def\vpl_warning#1{
    \out_llline{(SPECIAL~Warning:~#1)}
    \immediate\write16{Warning:~#1.}
}
```

\vpl_movert
\vpl_moveup

```
\def\vpl_movert#1{
    \afm_convert\a_dimen=#1;
    \out_llline{(MOVERIGHT~\vpl_real\a_dimen)}
}
\def\vpl_moveup#1{
    \afm_convert\a_dimen=#1;
    \out_llline{(MOVEUP~\vpl_real\a_dimen)}
}
```

\vpl_push
\vpl_pop

```
\def\vpl_push{\out_llline{(PUSH)}}
\def\vpl_pop{\out_llline{(POP)}}
```

12.8 Tidying up

\make_tidy

The only tidying up (currently) needed is to clear the list of mapfont numbers, since that is stored globally.

```
\def\make_tidy#1{
    \tidying_up_hook
    \if_including_map_
       \a_count=0\loop\ifnum\a_count<\font_count
          \edef\a_macro{\csname~f-\the\a_count\endcsname}
          \global\x_cs\let\a_macro\x_relax
          \advance\a_count by 1
       \repeat
       \global\font_count=0
    \fi
}
```

13 Font installation commands and .fd files

\installfonts
\endinstallfonts

The call

\installfonts ⟨*install commands*⟩ \endinstallfonts

is the top-level interface for installing a number of fonts and creating .fd files for them.

\installfamily
\installfont
\installrawfont

The ⟨*install commands*⟩ are:

\installfamily{⟨*encoding*⟩}{⟨*family*⟩}{⟨*FD-commmands*⟩}
\installfont{⟨*font-name*⟩}{⟨*file-list*⟩}{⟨*etx*⟩}
{⟨*encoding*⟩}{⟨*family*⟩}{⟨*series*⟩}{⟨*shape*⟩}{⟨*size*⟩}
\installrawfont{⟨*font-name*⟩}{⟨*file-list*⟩}{⟨*etx*⟩}
{⟨*encoding*⟩}{⟨*family*⟩}{⟨*series*⟩}{⟨*shape*⟩}{⟨*size*⟩}

Each \installfamily command causes the generation of an .fd file for ⟨*encoding*⟩ and ⟨*family*⟩, which is writen out by the time \endinstallfonts is processed.

Each \installfont generates a .vpl font by calling \etxtovpl and adds an .fd entry. Each \installrawfont generates a ligfull .pl font by calling \etxtopl and adds an .fd entry. (Raw .pl fonts, containing only the glyph metrics without any ligaturing or kerning information, are also generated by \mtxtopl called from \transformfont statements.)

\installfonts

Initializes the \family_toks token register, which is used to store the information which is written out to .fd files at the end of the job.

```
\newtoks\family_toks
\def\installfonts{
   \bgroup
   \global\family_toks={}
   \gdef\prev_file_list{}
}
```

\installfamily

\installfamily{⟨*encoding*⟩}{⟨*family*⟩}{⟨*FD-commmands*⟩}

Adds the command

\fd_family{⟨*encoding*⟩}{⟨*family*⟩}{⟨*FD-commmands*⟩}

\ENCODING-FAMILY

to the token list \family_toks and defines a macro \⟨*encoding*⟩-⟨*family*⟩.

```
\def\installfamily#1#2#3{
   \global\family_toks=
      \expandafter{\the\family_toks\fd_family{#1}{#2}{#3}}
   \global\x_cs\let{#1-#2}\empty_command
}
```

\installfont
\installrawfont

\installfont{⟨*font-name*⟩}{⟨*file-list*⟩}{⟨*etx*⟩}
{⟨*encoding*⟩}{⟨*family*⟩}{⟨*series*⟩}{⟨*shape*⟩}{⟨*size*⟩}
\installrawfont{⟨*font-name*⟩}{⟨*file-list*⟩}{⟨*etx*⟩}
{⟨*encoding*⟩}{⟨*family*⟩}{⟨*series*⟩}{⟨*shape*⟩}{⟨*size*⟩}

```
\def\installfont#1#2#3#4#5#6#7#8{
   \install_font{#1}{#2}{#3}{#4}{#5}{#6}{#7}{#8}{\etxtovpl}
}
```

```
\def\installrawfont#1#2#3#4#5#6#7#8{
   \record_usage{#1}
   \install_font{#1}{#2}{#3}{#4}{#5}{#6}{#7}{#8}{\etxtopl}
}
```

`\install_font` `\install_font{`⟨*font-name*⟩`}{`⟨*file-list*⟩`}{`⟨*etx*⟩`}`
`{`⟨*encoding*⟩`}{`⟨*family*⟩`}{`⟨*series*⟩`}{`⟨*shape*⟩`}{`⟨*size*⟩`}{`⟨*converter*⟩`}`

Installs a font shape by writing out a `.(v)pl` font using the given converter function and adding an entry to the `.fd` file.

```
\def\install_font#1#2#3#4#5#6#7#8#9{
   \xdef\curr_file_list{#2}
   \ifx\prev_file_list\curr_file_list\else
      \egroup\bgroup
      \expandafter\process_csep_list \expandafter\input_mtx_file
        \curr_file_list,\process_csep_list,
      \global\let\prev_file_list=\curr_file_list
   \fi
   #9{#3}{#1}
   \x_cs\ifx{#4-#5-#6-#7}\x_relax
      \let\do_shape=\x_relax
      \x_cs\xdef{#4-#5}{
         \x_cs\ifx{#4-#5}\x_relax\else
            \csname#4-#5\endcsname
         \fi
         \do_shape{#4}{#5}{#6}{#7}
      }
   \fi
   \let\do_size=\x_relax
   \x_cs\xdef{#4-#5-#6-#7}{
      \x_cs\ifx{#4-#5-#6-#7}\x_relax\else
         \csname#4-#5-#6-#7\endcsname
      \fi
      \do_size{#8}{#1}
   }
}
```

`\endinstallfonts` Finish the installation by processing the `\family_toks` token register, which contains the accumulated information to be written out to `.fd` files.

```
\def\endinstallfonts{
   \let\do_shape=\fd_shape
   \let\do_size=\fd_size
   \the\family_toks
   \global\family_toks{}
   \egroup
}
```

`\input_mtx_file` `\input_mtx_file{`⟨*file*⟩`}`

Process an `.mtx` file specified in the ⟨*file-list*⟩ argument of `\installfont` or `\installrawfont`. If an `.mtx` file doesn't exists, it is generated on the fly from a corresponding `.pl`, `.afm`, or `.vpl` file in the call to `\fromany`.

```
\def\input_mtx_file#1{
   \ifx\x_relax#1\x_relax\else
      \get_file_name #1~scaled~1000~;
```

```
      \fromany\file_name
      \ifisstr{transform-source}\then
         \inputmtx\file_name
      \fi
   \fi
}
```

\get_file_name The macro \get_file_name is expanded in the following context

\get_file_name ⟨*filename*⟩␣scaled␣⟨*scale*⟩␣⟨*junk*⟩ ;

```
\def\get_file_name #1~scaled~#2~#3;{
   \edef\file_name{#1}
   \ifnum#2=\one_thousand
      \unsetint{rawscale}
   \else
      \x_resetint{rawscale}{#2}
   \fi
}
```

13.1 Writing font definition files

\fd_family \fd_family{ENCODING}{FAMILY}{FD-COMMANDS}

Writes out an .fd file for the specified ENCODING and FAMILY processing the accumulated information and default substitutions.

```
\def\fd_family#1#2#3{
   \a_toks{#3}
   \edef\lowercase_file{\lowercase{
     \edef\noexpand\lowercase_file{#1#2.fd}}}
   \lowercase_file
   \open_out{\lowercase_file}
   \out_line{\percent_char~Filename:~\lowercase_file}
   \out_line{\percent_char~Created~by:~tex~\jobname}
   \out_line{\percent_char~Created~using~fontinst~v\fontinstversion}
   \out_line{}
   \out_line{\percent_char~THIS~FILE~SHOULD~BE~PUT~IN~A~TEX~INPUTS~
      DIRECTORY}
   \out_line{}
   \out_line{\string\ProvidesFile{\lowercase_file}}
   \out_lline{[
      \the\year/
      \ifnum10>\month0\fi\the\month/
      \ifnum10>\day0\fi\the\day\space
      Fontinst~v\fontinstversion\space
      font~definitions~for~#1/#2.
   ]}
   \out_line{}
   \out_line{\string\DeclareFontFamily{#1}{#2}{\the\a_toks}}
   {
      \csname #1-#2\endcsname
      \out_line{}
      \let\do_shape=\substitute_shape
      \csname #1-#2\endcsname
      \let\do_shape=\remove_shape
```

```
      \csname #1-#2\endcsname
   }
   \x_cs\g_let{#1-#2}\x_relax
   \out_line{}
   \out_line{\string\endinput}
   \close_out{Font~definitions}
}
```

`\fd_shape` `\fd_shape{ENCODING}{FAMILY}{SERIES}{SHAPE}`

```
\def\fd_shape#1#2#3#4{
   \out_line{}
   \out_line{\string\DeclareFontShape{#1}{#2}{#3}{#4}\left_brace_char}
   \csname #1-#2-#3-#4\endcsname
   \x_cs\g_let{#1-#2-#3-#4}\empty_command
   \out_line{\right_brace_char{}}
}
```

`\fd_size` `\fd_size{SIZE}{FONT-NAME}`

```
\def\fd_size#1#2{
   \x_cs\ifx{siz-#1}\x_relax
      \out_lline{#1~#2}
   \else
      \out_lline{\csname siz-#1\endcsname\space #2}
   \fi
}
```

`\remove_shape` `\remove_shape{ENCODING}{FAMILY}{SERIES}{SHAPE}`

```
\def\remove_shape#1#2#3#4{
   \x_cs\g_let{#1-#2-#3-#4}\x_relax
}
```

`\substitute_shape` `\substitute_shape{ENCODING}{FAMILY}{SERIES}{SHAPE}`

```
\def\substitute_shape#1#2#3#4{
   \edef\orig_shape{#4}
   \substitute_series{#1}{#2}{#3}{\orig_shape}
   \x_cs\ifx{sub-\orig_shape}\x_relax\else
      \edef\subst_shape{\csname sub-\orig_shape\endcsname}
      \x_cs\ifx{#1-#2-#3-\subst_shape}\x_relax
         \out_line{
            \string\DeclareFontShape{#1}{#2}{#3}{\subst_shape}{
               <->\csname typ-\orig_shape\endcsname\space
               *~#2/#3/\orig_shape
            }{}
         }
         \x_cs\let{#1-#2-#3-\subst_shape}\empty_command
         \substitute_shape{#1}{#2}{#3}{\subst_shape}
      \fi
   \fi
}
```

`\substitute_series` `\substitute_series{ENCODING}{FAMILY}{SERIES}{SHAPE}`

```
\def\substitute_series#1#2#3#4{
   \edef\orig_series{#3}
```

```
    \x_cs\ifx{sub-\orig_series}\x_relax\else
       \edef\subst_series{\csname sub-\orig_series\endcsname}
       \x_cs\ifx{#1-#2-\subst_series-#4}\x_relax
          \out_line{
             \string\DeclareFontShape{#1}{#2}{\subst_series}{#4}{
                <->\csname typ-\orig_series\endcsname\space
                *~#2/\orig_series/#4
             }{}
          }
          \x_cs\let{#1-#2-\subst_series-#4}\empty_command
          \substitute_series{#1}{#2}{\subst_series}{#4}
       \fi
    \fi
}
```

`\substitutesilent` `\substitutesilent{TO}{FROM}`
`\substitutenoisy` `\substitutenoisy{TO}{FROM}`

Specifies a default substitution for family or shape `TO`, which is substituted by family or shape `FROM`.

```
\def\substitutesilent#1#2{
   \x_cs\def{sub-#2}{#1}
   \x_cs\def{typ-#2}{ssub}
}
\def\substitutenoisy#1#2{
   \x_cs\def{sub-#2}{#1}
   \x_cs\def{typ-#2}{sub}
}

\substitutesilent{bx}{b}
\substitutesilent{b}{bx}
\substitutesilent{b}{sb}
\substitutesilent{b}{db}
\substitutesilent{m}{mb}
\substitutesilent{m}{l}

\substitutenoisy{ui}{it}
```

I don't think we want these since in OT1 encoding it will cause the old `<sterling>` vs `<dollar>` problem. (ASAJ) — Oh yes we do. (SPQR)

```
\substitutesilent{sl}{it}
\substitutesilent{it}{sl}
```

13.2 Default encodings and font sizes

`\declareencoding` `\declareencoding{CODINGSCHEME}{ENCODING}`

Declare a macro `\enc-CODINGSCHEME` which expands to `ENCODING`. This is used to determine the encoding in `\pltomtx`.

```
\def\declareencoding#1#2{\x_cs\edef{enc-#1}{#2}}
```

Old TEX text font encodings.

```
\declareencoding{TEX~TEXT}{OT1}
\declareencoding{TEX~TEXT~WITHOUT~F-LIGATURES}{OT1}
\declareencoding{TEX~TYPEWRITER~TEXT}{OT1TT}
```

Old TEX math font encodings.

```
\declareencoding{TEX~MATH~ITALIC}{OML}
\declareencoding{TEX~MATH~SYMBOLS}{OMS}
\declareencoding{TEX~MATH~EXTENSION}{OMX}
\declareencoding{LATEX~SYMBOLS}{lasy}
```

Euler math font encodings.

```
\declareencoding{TEX~TEXT~SUBSET}{eufrak}
\declareencoding{TEX~MATH~ITALIC~SUBSET}{eurm}
\declareencoding{TEX~MATH~SYMBOLS~SUBSET}{euscr}
\declareencoding{EULER~SUBSTITUTIONS~ONLY}{euex}
```

New TEX text font encodings.

```
\declareencoding{EXTENDED~TEX~FONT~ENCODING~-~LATIN}{T1}
\declareencoding{TEX~TEXT~COMPANION~SYMBOLS~1---TS1}{TS1}
```

Rencoded PostScript font encdings.

```
\declareencoding{TEXBASE1ENCODING}{8r}
\declareencoding{TEX~TYPEWRITER~AND~WINDOWS~ANSI}{8y}
```

`\declaresize` `\declaresize{FONTSIZE}{LATEXSIZE}`

```
\def\declaresize#1#2{\x_cs\edef{siz-#1}{#2}}
```

Default sizes. An empty size argument is equivalent to `<->`, for use with scalable fonts.

```
\declaresize{}{<->}
\declaresize{5}{<5>}
\declaresize{6}{<6>}
\declaresize{7}{<7>}
\declaresize{8}{<8>}
\declaresize{9}{<9>}
\declaresize{10}{<10>}
\declaresize{11}{<10.95>}
\declaresize{12}{<12>}
\declaresize{14}{<14.4>}
\declaresize{17}{<17.28>}
\declaresize{20}{<20.74>}
\declaresize{25}{<24.88>}
```

14 Debugging

`\NOFILES` `\NOFILES` switches off file generation, and causes fontinst to only generate empty files.

```
\def\NOFILES{
   \def\transformfont##1##2{
      \touch_file{##1.mtx}
      \touch_file{##1.pl}
   }
   \def\installfonts{}
   \def\endinstallfonts{}
   \def\installfont##1##2##3##4##5##6##7##8{%
      \touch_file{##1.vpl}
   }
```

```
    \def\installrawfont##1##2##3##4##5##6##7##8{%
       \touch_file{##1.pl}
    }
    \def\installfamily##1##2##3{
       \touch_file{##1##2.fd}
    }
}
```

`\touch_file`

```
\def\touch_file#1{
    \edef\lowercase_file{\lowercase{
      \edef\noexpand\lowercase_file{#1}}}
    \lowercase_file
    \open_out{\lowercase_file}
    \out_line{\percent_char~TEST~FILE.}
    \out_line{\percent_char~Created~whilst~debugging~fontinst.}
    \close_out{Test~file}
}
⟨/pkg⟩
```

Set `\newlinechar` for `\errhelp` messages.

```
⟨∗pkg | misc⟩
\newlinechar=‘\^^J
```

By default, show as much error info as you can. (I assume fontinst users are TEXperts.)

```
\errorcontextlines=999
⟨/pkg | misc⟩
```

File c
ficonv.dtx

15 Basic file format conversions

15.1 Converting an ENC file to an ETX file

\enctoext The macro

\enctoetx{⟨*encfile*⟩}{⟨*etxfile*⟩}

reads ⟨*encfile*⟩.enc and writes the same information to ⟨*etxfile*⟩.etx, in a format TEX can read more easily.

Each /⟨*glyph*⟩ command is recorded in a macro \o-⟨*glyph*⟩, which expands to a corresponding \setslot{⟨*glyph*⟩} ... \endslot statement.

```
⟨*misc⟩
\newif\ifmissingslots
\x_cs\def{o-.notdef}#1{\global\missingslotstrue}
```

\enctoext

```
{
   \catcode`\/=\active
   \catcode`\]=\active
   \gdef\enctoetx#1#2{{
      \catcode`\/=\active
      \catcode`\]=\active
      \def/##1[{
         \a_count=0
         \global\missingslotsfalse
         \def/####1~{
            \csname~o-####1\endcsname{
               \ifmissingslots
                  \out_line{\string\nextslot{\the\a_count}}
               \fi
               \global\missingslotsfalse
               \out_line{\string\setslot{####1}}
               \out_line{\string\endsetslot}
               \out_line{}
            }
            \advance\a_count by 1
         }
      }
      \def]~def{}
      \make_etx{#1}{#2}
   }}
}
```

\make_etx finishes the job of \enctoext.

```
\def\make_etx#1#2{
   \open_out{\temp_prefix#2.etx}
   \out_line{\percent_char~Filename:~#2.etx}
   \out_line{\percent_char~Created~by:~tex~\jobname}
```

```
    \out_line{\percent_char~Created~using:~\string\enctoetx{#1}{#2}}
    \out_line{}
    \out_line{\percent_char~This~file~contains~the
       information~of~#1.enc~in~a~form}
    \out_line{\percent_char~more~easily~read~by~TeX.
       It~is~used~by~the~fontinst~package.}
    \out_line{}
    \out_line{\percent_char~THIS~FILE~CAN~BE~DELETED.}
    \out_line{}
    \out_line{\string\relax}
    \out_line{}
    \out_line{\string\documentclass[twocolumn]{article}}
    \out_line{\string\usepackage{fontdoc}}
    \out_line{}
    \out_line{\string\begin{document}}
    \out_line{}
    \out_line{This~document~describes~the~#1~encoding.}
    \out_line{It~was~automatically~generated~by~the}
    \out_line{{\string\tt\space~fontinst}~package.}
    \out_line{}
    \out_line{\string\encoding}
    \out_line{}
    \out_line{\string\needsfontinstversion{\fontinstversion}}
    \out_line{}
    \primitiveinput #1.enc\x_relax
    \out_line{}
    \out_line{\string\end encoding}
    \out_line{}
    \out_line{\string\end{document}}
    \close_out{Encoding~vector}
}
⟨/misc⟩
```

15.2 Converting an AFM file to an MTX file

`\afmtomtx` The macro

`\afmtomtx{`⟨*afmfile*⟩`}{`⟨*mtxfile*⟩`}`

reads ⟨*afmfile*⟩`.afm`, and writes the same information out to ⟨*mtxfile*⟩`.mtx`.

`\afmtomtx`

```
⟨*pkg⟩
\def\afmtomtx#1#2{{
    \open_out{\temp_prefix#2.mtx}
    \def\raw_font_name{#2}
    \x_resetint{italicslant}{0}
    \let\italcorr_expression=\uprightitalcorr
    \x_setint{minimumkern}{0}
    \minimum_kern=\int{minimumkern}
    \global\let\afm_font_name=\empty_command
    \out_line{\percent_char~Filename:~#2.mtx}
    \out_line{\percent_char~Created~by:~tex~\jobname}
    \out_line{\percent_char~Created~using:~\string\afmtomtx{#1}{#2}}
```

```
   \out_line{}
   \out_line{\percent_char~This~file~contains~the
      information~of~#1.afm~in~a~form}
   \out_line{\percent_char~more~easily~read~by~TeX.~
      It~is~used~by~the~fontinst~package.}
   \out_line{}
   \out_line{\percent_char~THIS~FILE~CAN~BE~DELETED.}
   \out_line{}
   \out_line{\string\relax}
   \out_line{\string\metrics}
   \out_line{}
   \out_line{\string\needsfontinstversion{\fontinstversion}}
   \out_line{}
   \catcode`\^^M=12
   \catcode`\ =10
   \expandafter\afm_line\primitiveinput #1.afm\x_relax
   \out_line{}
   \out_line{\endmetrics_text}
   \close_out{Metrics}
}}
```

Kerns below this value are ignored.

```
\newcount\minimum_kern
```

`\afm_length` `\afm_unit_dimen`

The call `\afm_length`⟨*count*⟩`{`⟨*real*⟩`}` interprets the ⟨*real*⟩ as a real number, rounds it to the nearest integer, and sets the ⟨*count*⟩ (a `\count` register) to that integer. In this process, `\a_dimen` is used as a temporary storage.

```
\def\afm_length#1#2{
   \a_dimen=#2\afm_unit_dimen
   #1=\a_dimen
   \divide #1 by \afm_unit_dimen
   \advance \a_dimen by -#1\afm_unit_dimen
   \ifdim \a_dimen>0.5\afm_unit_dimen
      \advance #1 by 1
   \else \ifdim \a_dimen<-0.5\afm_unit_dimen
      \advance #1 by -1
   \fi\fi
   \x_relax
}
```

The dimen `\afm_unit_dimen` is used to keep track of how long an AFM unit is interpreted as being in this routine. Lowering its value makes `\afm_length` capable of handling greater lengths but looses some very slight precision in the rounding, increasing the value has the opposite effects. The current value of 1000 sp means it reads lengths with three decimals accuracy (not very much use for them though as the number is rounded to zero decimals accuracy anyway, but it does make a difference when deciding how a ⟨*real*⟩ like `0.502` should be rounded) and can handle lengths of an absolute value of a good million AFM units. This should be adequate in most cases. It is, by the way, probably wisest to keep it a power of ten scaled points in all cases, as this should reduce the rounding errors caused by various base conversions.

```
\newdimen\afm_unit_dimen
\afm_unit_dimen=1000sp
```

The command `\afm_line` reads to the end of the line, calls `\afm_command` on that line, then calls `\afm_line` again.

```
{\catcode`\^^M=12 \gdef\afm_line#1
{\afm_command#1~\end_of_line\afm_line}}
```

The command `\afm_command` reads the first word `FOO`, and calls `afm-FOO`. If this does not exist, then `\gobble_one_line` will eat up the rest of the line.

```
\def\afm_command#1~{\csname~afm-#1\endcsname\gobble_one_line}
\def\gobble_one_line#1\end_of_line{}
```

This all stops when we reach the command `EndFontMetrics`.

```
\x_cs\def{afm-EndFontMetrics}#1\afm_line{\endinput}
```

`\afm_def` To define an AFM command, you say `\afm_def{`⟨*command*⟩`}(`⟨*pattern*⟩`){`⟨*result*⟩`}`

```
\def\afm_def#1(#2)#3{\x_cs\def{afm-#1}
   \gobble_one_line#2\end_of_line{#3}}
```

`\afm_let` Saying `\afm_let{`⟨*dest-command*⟩`}{`⟨*source-command*⟩`}` copies the definition of one AFM command to another.

```
\def\afm_let#1#2{
   \expandafter\let \csname afm-#1\expandafter\endcsname
      \csname afm-#2\endcsname
}
```

For example, we can define the following AFM commands:

```
\afm_def{CharWidth}(#1){\afm_length\char_x_width{#1}}
\afm_def{ItalicAngle}(#1){\calculate_it_slant{#1}}
\afm_def{XHeight}(#1){
   \afm_length\a_count{#1}
   \out_line{\string\setint{xheight}{\the\a_count}}}
\afm_def{CapHeight}(#1){
   \afm_length\a_count{#1}
   \out_line{\string\setint{capheight}{\the\a_count}}}
\afm_def{Ascender}(#1){
   \afm_length\a_count{#1}
   \out_line{\string\setint{ascender}{\the\a_count}}}
\afm_def{Descender}(#1){
   \afm_length\a_count{#1}
   \out_line{\string\setint{descender_neg}{\the\a_count}}}
\afm_def{UnderlineThickness}(#1){
   \afm_length\a_count{#1}
   \out_line{\string\setint{underlinethickness}{\the\a_count}}}
\afm_def{FontBBox}(#1~#2~#3~#4){
   \afm_length\a_count{#4}
   \out_line{\string\setint{maxheight}{\the\a_count}}
   \afm_length\a_count{#2}
   \out_line{\string\setint{maxdepth_neg}{\the\a_count}}}
\afm_def{IsFixedPitch}(#1){
   \if\first_char#1=f
   \else\out_line{\string\setint{monowidth}{1}}
   \fi
}
```

\afm_font_name The FontName of a font is recorded in the macro \afm_font_name. This information is of no use when making TFMs and VFs, but it is likely to be of use for generation of map files, so it will be included in a file of recorded transforms, if such a file is being generated.

```
\afm_def{FontName}(#1~){\xdef\afm_font_name{#1}}
```

Processing kern pairs. If one of the glyph name starts with a dot as in .notdef or .null the kern pair is ignored.

```
\afm_def{KP}(#1~#2~#3~#4){
   \if\first_char#1=.\else
   \if\first_char#2=.\else
      \afm_length\a_count{#3}
      \ifnum \a_count>\minimum_kern
         \out_line{\string\setkern{#1}{#2}{\the\a_count}}
      \else\ifnum \a_count<-\minimum_kern
         \out_line{\string\setkern{#1}{#2}{\the\a_count}}
      \fi\fi
   \fi\fi
}
\afm_let{KPX}{KP}
```

Processing char metrics.

```
\afm_def{C}(#1~;#2){\init_afm{#1}\do_list[#2]\afm_char}
\afm_let{CH}{C}
```

Processing composite chars.

```
\afm_def{CC}(#1~#2~;#3){\init_cc{#1}\do_list[#3]\cc_char}
```

When parsing a character, we set the values of the following variables:

```
\newcount\char_slot
\newcount\char_x_width
\newcount\x_width
\newcount\bbox_llx
\newcount\bbox_lly
\newcount\bbox_urx
\newcount\bbox_ury
\let\char_name=\empty_command
```

\init_afm initializes the variables the AFM character list writes to.

```
\def\init_afm#1{
   \char_slot=#1\x_relax
   \x_width=\char_x_width
   \bbox_llx=0
   \bbox_lly=0
   \bbox_urx=0
   \bbox_ury=0
   \let\char_name=\empty_command
}
\def\afm_char{
   \a_count=-\bbox_lly
   \eval_expr{
      \italcorr_expression\x_width\bbox_llx\bbox_urx\bbox_lly\bbox_ury
   }
```

```
    \out_line{
       \ifnum -1<\char_slot
          \string\setrawglyph
       \else
          \string\setnotglyph
       \fi
       {\char_name}
       {\raw_font_name}
       {10pt}
       {\the\char_slot}
       {\the\x_width}
       {\the\bbox_ury}
       {\the\a_count}
       {\the\result}
    }
}
```

`\init_cc` and `\cc_char` write out a composite character glyph.

```
\def\init_cc#1{%
    \out_line{\string\setglyph{#1}}
    \def\char_name{#1}
}
\def\cc_char{%
    \out_lline{\string\samesize{\char_name-not}}
    \out_line{\string\endsetglyph}
}
```

`\italcorr_expression`

The way the italic correction is computed has been changed quite a bit, although the computed values are still the same. The point is that it is much simpler to modify the formula according to which the value is computed using this method than using the previous method.

The call

`\italcorr_expression{`⟨*width*⟩`}{`⟨*left*⟩`}{`⟨*right*⟩`}{`⟨*bottom*⟩`}{`⟨*top*⟩`}`,

where the arguments are TEX ⟨*number*⟩s, should expand to an integer expression. The value of that expression will be taken as the italic correction of the current character.

⟨*width*⟩ is the width of the character. ⟨*left*⟩, ⟨*right*⟩, ⟨*bottom*⟩, and ⟨*top*⟩ are the respective coordinates of the sides of the bounding box of the character. A quantity which is not given as an argument, but which nontheless might be of interest for a calculation of italic correction, is the italic slant of the font. This quantity can be found in the fontinst integer `italicslant`. (The MTX file written will also set the integer `italicslant` to this value.)

`\uprightitalcorr` `\slanteditalcorr`

These two commands are what `\italcorr_expression` will get set to—the slanted version is used if the italic slant is positive and the upright version is used otherwise. The default definitions compute the same values as in fontinst v 1.8, but the definitions can easily be modified using `\resetcommand`.

```
\def\uprightitalcorr#1#2#3#4#5{0}
\def\slanteditalcorr#1#2#3#4#5{\max{\sub{#3}{#1}}{0}}
```

To set the italic angle, we need to calculate the tangent of the angle that the `.afm` file contains. This is done with David Carlisle's trig macros. We need to do

a bit of trickery to strip off any spaces which may have crept into the end of the angle, since the trig macros don't like space at the end of their argument.

? `\afm_line` **always inserts a space at the end of the line.** Could we therefore save us a bit of trouble by simply changing the pattern for `\ItalicAngle` to `(#1~)`? /LH

```
\def\calculate_it_slant#1{
   \edef\theangle{\strip_spaces#1~\end_strip_spaces}
   \CalculateTan{\theangle}
   \a_dimen=-\one_thousand sp
   \a_dimen=\UseTan{\theangle}\a_dimen
   \a_count=\a_dimen
   \out_line{\string\setint{italicslant}{\the\a_count}}
   \x_resetint{italicslant}{\a_count}
   \ifnum 0<\a_count
      \let\italcorr_expression=\slanteditalcorr
   \else
      \let\italcorr_expression=\uprightitalcorr
   \fi
}
\def\strip_spaces#1~#2\end_strip_spaces{#1}
```

To process a list of commands separated by semi-colons, we call `\do_list[LIST]`. This works in a similar way to `\afm_line`.

```
\def\do_list[~#1~#2;~#3]{
   \csname~list-#1\endcsname\gobble_one_semi#2;
   \ifx\x_relax#3\x_relax\expandafter\gobble_one
   \else\expandafter\identity_one\fi
   {\do_list[~#3]}
}
\def\gobble_one_semi#1;{}
```

There is an analagous `\list_def` for defining commands to be used inside lists.

```
\def\list_def#1(#2)#3{\x_cs\def{list-#1}\gobble_one_semi#2~;{#3}}
```

For example, these are the commands that are used in giving character metrics:

```
\list_def{W}(#1~#2){\afm_length\x_width{#1}}
\list_def{WX}(#1){\afm_length\x_width{#1}}
\list_def{WY}(#1){}
\list_def{N}(#1){\def\char_name{#1}}
\list_def{B}(#1~#2~#3~#4){
   \afm_length\bbox_llx{#1}
   \afm_length\bbox_lly{#2}
   \afm_length\bbox_urx{#3}
   \afm_length\bbox_ury{#4}
}
\list_def{PCC}(#1~#2~#3){
   \afm_length\a_count{#2}
   \afm_length\b_count{#3}
   \out_lline{\string\glyphpcc{#1}{\the\a_count}{\the\b_count}}
}
```

15.3 Converting a PL file to an MTX file

\generalpltomtx The macro

\generalpltomtx{⟨*plfile*⟩}{⟨*mtxfile*⟩}{⟨*plsuffix*⟩}{⟨*opt-enc*⟩}

reads ⟨*plfile*⟩.⟨*plsuffix*⟩, interprets it as having the encoding specified by the file ⟨*opt-enc*⟩.etx, and writes the same metric information out to ⟨*mtxfile*⟩.mtx. In case ⟨*opt-enc*⟩ is empty, the encoding will be determined using the CODINGSCHEME

\pltomtx property of the file being read. The macro

\pltomtx{⟨*plfile*⟩}{⟨*mtxfile*⟩}

reads ⟨*plfile*⟩.pl, uses the CODINGSCHEME property in that file to determine its encoding, and writes the same metric information out to ⟨*mtxfile*⟩.mtx.

None of these commands can cope with SKIP commands.

\generalpltomtx

```
\def\generalpltomtx#1#2#3#4{{
   \ifx _#4_\else
      \def\do_slot{\x_cs\let{name-\the\slot_number}\slot_name}
      \def\do_boundary{\x_cs\let{name-BOUNDARYCHAR}\slot_name}
      \inputetx{#4}
      \let\CODINGSCHEME=\ignore_parens
   \fi
   \pl_to_mtx{#1}{#2}{#3}{\string\generalpltomtx{#1}{#2}{#3}{#4}}
}}
```

\pltomtx

```
\def\pltomtx#1#2{\generalpltomtx{#1}{#2}{pl}{}}
```

\pl_to_mtx The \pl_to_mtx macro contains all code that was common to \pltomtx and \generalpltomtx before the former was redefined to a call of the latter. The structure of a call of \pl_to_mtx is

\pl_to_mtx{⟨*plfile*⟩}{⟨*mtxfile*⟩}{⟨*plsuffix*⟩}{⟨*call*⟩}

⟨*call*⟩ is what should be written in the "Created using:" comment at the top of the MTX file written.

```
\def\pl_to_mtx#1#2#3#4{
   \def\raw_font_name{#1}
   \open_out{\temp_prefix#2.mtx}
   \out_line{\percent_char~Filename:~#2.mtx}
   \out_line{\percent_char~Created~by:~tex~\jobname}
   \out_line{\percent_char~Created~using:~#4}
   \out_line{}
   \out_line{\percent_char~This~file~contains~the~
      information~of~#1.#3~in~a~form}
   \out_line{\percent_char~more~easily~read~by~TeX.~
      It~is~used~by~the~fontinst~package.}
   \out_line{}
   \out_line{\percent_char~THIS~FILE~CAN~BE~DELETED.}
   \out_line{}
   \out_line{\string\relax}
```

```
   \out_line{\string\metrics}
   \out_line{}
   \out_line{\string\needsfontinstversion{\fontinstversion}}
   \out_line{}
   \catcode`\(=0 \catcode`\)=9
   \let\/=\ignore_parens
   \let\do_pl_glyph=\x_relax
   \primitiveinput #1.#3\x_relax
   \do_pl_glyph
   \out_line{}
   \ifisint{\percent_char boundarychar}\then
      \f_count=\int{\percent_char boundarychar}
      \x_cs\ifx{name-\the\f_count}\x_relax \else
         \out_line{\string\setstr{rightboundary}
            {\csname name-\the\f_count\endcsname}
         }
         \out_line{}
      \fi
   \fi
   \out_line{\endmetrics_text}
   \close_out{Metrics}
}
```

To parse a `.pl` file, we first make `(` the escape character, make `)` ignored, then define the various PL commands.

We can ignore a parenthesis matched string by making `(` and `)` the group delimiters, then gobbling them up.

```
\def\ignore_parens{\bgroup\catcode`(=1 \catcode`)=2 \x_relax
   \expandafter\expandafter\expandafter\gobble_parens
      \iftrue\expandafter{\else}\fi}
\def\gobble_parens#1{\egroup}
```

Convert a PL real to an AFM unit, assuming it contains a decimal point.

A **`\pl_real` only works if the `DESIGNUNITS` setting is at the default value 1.** Luckily, this is the case with the Computer Modern fonts (at least the ones I looked at), so it works for those. What is not so luckily is that it fails quite spectacularly for the VPL files fontinst generates itself—usually everything comes out ten times as large as it should be. Is there a reasonable way around this? /LH

```
\def\pl_real#1{\pl_realer(#1000)}
\def\pl_realer(#1.#2#3#4#5){#1#2#3#4}
```

Convert a PL int to a TEX int, assuming it's prefixed by `C`, `D`, `O`, or `H`.

```
\def\pl_int#1#2{
   \ifx#1C `#2
   \else\ifx#1D #2
   \else\ifx#1O '#2
   \else\ifx#1H "#2
   \else -1\errmessage{Unknown~PL~number~prefix~`#1'}
   \fi\fi\fi\fi
}
```

Many of the PL commands are ignored, and I'm assuming the `R`s are in the places `tftopl` puts them, which is a bit naughty of me.

```
\let\COMMENT=\ignore_parens
\let\FAMILY=\ignore_parens
\let\FACE=\ignore_parens
\let\CHECKSUM=\ignore_parens
\let\LIG=\ignore_parens
\let\NEXTLARGER=\ignore_parens
\let\VARCHAR=\ignore_parens
```

The properties which are unique for VPL files—VTITLE, MAPFONT, MAP, FONTNAME, FONTAREA, FONTCHECKSUM, FONTAT, FONTDSIZE, SELECTFONT, SETCHAR, SETRULE, MOVERIGHT, MOVELEFT, MOVEUP, MOVEDOWN, PUSH, POP, SPECIAL, and SPECIALHEX—should also be ignored, but it is actually sufficient to ignore the first three since the others are only allowed inside MAP or MAPFONT property lists.

```
\let\VTITLE=\ignore_parens
\let\MAPFONT=\ignore_parens
\let\MAP=\ignore_parens
```

When we reach a CODINGSCHEME instruction, we read the coding string, and read in the corresponding ⟨*encoding*⟩`.etx` file.

The corresponding ⟨*encoding*⟩ is specified by `\declareencoding` statements (see below). Each `\declare_encoding` defines a macro `\enc-`⟨*codingscheme*⟩ which expands to ⟨*encoding*⟩.

If the PL file is converted using the `\generalpltomtx` command with a nonempty ⟨*opt-enc*⟩ argument then the CODINGSCHEME instruction is ignored since an encoding file has already been read in.

```
\def\CODINGSCHEME{\bgroup\catcode`\)=12\x_relax\CODINGSCHEME_cont}
\def\CODINGSCHEME_cont#1){
   \egroup
   \x_cs\ifx{enc-#1}\x_relax
      \errhelp{The~encoding~`#1'~has~not~been~declared.^^J
         You~should~declare~it~with~
         \string\declareencoding{#1}{ETXFILE}.^^J
         Press~<RETURN>~to~carry~on~with~fingers~crossed,^^J
         or~X~<RETURN>~to~exit.}
      \errmessage{Undeclared~encoding~`#1'}
   \else
      \def\do_slot{\x_cs\let{name-\the\slot_number}\slot_name}
      \def\do_boundary{\x_cs\let{name-BOUNDARYCHAR}\slot_name}
      \catcode`\(=12 \catcode`\)=12
      \x_cs\inputetx{enc-#1}\x_relax
      \catcode`\(=0 \catcode`\)=9
   \fi
}
```

```
\def\DESIGNSIZE~#1~#2~{
   \a_dimen=#2pt
   \out_line{\string\setdim{designsize}{\the\a_dimen}}
}
```

`\DESIGNUNITS` The PL to MTX converter assumes that the (V)PL files to convert look like the ones created by TFtoPL/VFtoVP, and the interpretation of the DESIGNUNITS property is one thing specifically affected by this. The TFM file format does not store the DESIGNUNITS value used, so the two above programs always generate

(V)PL files with the default setting of design unit equal to the design size. Hence any occurence of the DESIGNUNITS property is an indication of an error.

The incorrect metrics can be corrected by scaling by a suitable amount (1000 divided by the designunits dimen), but it is much simpler to convert the PL to a TFM and then convert it back, that will also fix the units.

```
\def\DESIGNUNITS~#1~#2~{
   \a_dimen=#2pt
   \ifdim 1pt=\a_dimen \else
      \fontinsterror{PL-to-MTX}{Nondefault~unit~used~in~PL~file}
         {You~may~continue,~but~the~metrics~for~this~font~will~be~wrong.}
   \fi
   \out_line{\string\setdim{designunits}{\the\a_dimen}}
}

\def\BOUNDARYCHAR~#1~#2~{
   \x_setint{\percent_char boundarychar}{\pl_int{#1}{#2}}
}
```

The following fontdimens are converted to ints and written out to the .mtx file.

```
\let\FONTDIMEN=\x_relax
\def\SLANT~R~#1~{\out_line{\string\setint{italicslant}{\pl_real{#1}}}}
\def\SPACE~R~#1~{\out_line{\string\setint{interword}{\pl_real{#1}}}}
\def\STRETCH~R~#1~{\out_line{\string\setint{stretchword}{\pl_real{#1}}}}
\def\SHRINK~R~#1~{\out_line{\string\setint{shrinkword}{\pl_real{#1}}}}
\def\XHEIGHT~R~#1~{\out_line{\string\setint{xheight}{\pl_real{#1}}}}
\def\QUAD~R~#1~{\out_line{\string\setint{quad}{\pl_real{#1}}}}
\def\EXTRASPACE~R~#1~{
   \out_line{\string\setint{extraspace}{\pl_real{#1}}}
}
\def\DEFAULTRULETHICKNESS~R~#1~{
   \out_line{\string\setint{underlinethickness}{\pl_real{#1}}}
}
```

The following fontdimens are ignored.

```
\def\HEADER~#1~#2~#3~#4~{}
\def\SEVENBITSAFEFLAG~#1~{}
\def\PARAMETER~#1~#2~#3~#4~{}
\def\NUM#1~#2~#3~{}
\def\DENOM#1~#2~#3~{}
\def\SUP#1~#2~#3~{}
\def\SUB#1~#2~#3~{}
\def\SUBDROP~#1~#2~{}
\def\SUPDROP~#1~#2~{}
\def\DELIM#1~#2~#3~{}
\def\AXISHEIGHT~#1~#2~{}
\def\BIGOPSPACING#1~#2~#3~{}
```

The following ligtable commands are processed.

```
\def\LIGTABLE{\let\do=\never_do\let\a_macro\empty_command}
\def\LABEL~#1{\ifx #1B
      \expandafter\LABEL_boundarychar
   \else
      \expandafter\LABEL_slot \expandafter#1
```

```
   \fi
}
\def\LABEL_slot #1~#2~{
   \f_count=\pl_int{#1}{#2}
   \edef\a_macro{\a_macro
      \x_cs\do_if_defined{name-\the\f_count}
   }
}
\def\LABEL_boundarychar OUNDARYCHAR{
   \edef\a_macro{\a_macro
      \x_cs\do_if_defined{name-BOUNDARYCHAR}
   }
}
\def\do_if_defined#1{\ifx #1\x_relax \else \do{#1} \fi}
\def\STOP{\let\a_macro\empty_command}
\def\SKIP~#1~#2~{\immediate\write16{Warning:~SKIP~instruction~ignored.}}
\def\KRN~#1~#2~R~#3~{
   \edef\do{\noexpand\write_pl_krn{\pl_int{#1}{#2}}{\pl_real{#3}}}
   \a_macro
   \let\do=\never_do
}
\def\write_pl_krn#1#2#3{
   \f_count=#1\x_relax
   \x_cs\ifx{name-\the\f_count}\x_relax \else
      \out_line{\string\setkern{#3}
         {\csname name-\the\f_count\endcsname}{#2}
      }
   \fi
}
```

The following character metrics are processed.

```
\def\CHARWD~R~#1~{\b_count=\pl_real{#1}}
\def\CHARHT~R~#1~{\c_count=\pl_real{#1}}
\def\CHARDP~R~#1~{\d_count=\pl_real{#1}}
\def\CHARIC~R~#1~{\e_count=\pl_real{#1}}
\def\CHARACTER~#1~#2~{
   \do_pl_glyph
   \a_count=\pl_int{#1}{#2}
   \b_count=0
   \c_count=0
   \d_count=0
   \e_count=0
   \let\do_pl_glyph=\write_pl_glyph
}
\def\write_pl_glyph{
      \x_cs\ifx{name-\the\a_count}\x_relax\else
         \out_line{\string\setrawglyph
            {\csname~name-\the\a_count\endcsname}
            {\raw_font_name}
            {\the\a_dimen}
            {\the\a_count}
            {\the\b_count}
            {\the\c_count}
```

```
            {\the\d_count}
            {\the\e_count}}
      \fi
  }
```

15.4 Converting an MTX file to a PL file

`\mtxtopl` The macro

`\mtxtopl{`⟨*mtxfile*⟩`}{`⟨*plfile*⟩`}`

writes a font from the `\setrawglyph` instructions in ⟨*mtxfile*⟩ to ⟨*plfile*⟩. It ignores any font dimensions and kerning, so the resulting font is only useful for generating virtual fonts from. (This macro is called by `\transformfont`.)

`\mtxtopl`

```
\def\mtxtopl#1#2{{
  \open_out{#2.pl}
  \out_line{(COMMENT~raw~font~#2~created~by~fontinst~
     v\fontinstversion)}
  \out_line{}
  \out_line{(COMMENT~Filename:~#2.pl)}
  \out_line{(COMMENT~Created~by:~tex~\jobname)}
  \out_line{(COMMENT~Created~using:~\string\mtxtopl{#1}{#2})}
  \out_line{}
  \out_line{(COMMENT~This~file~can~be~turned~into~a~TeX~font~with)}
  \out_line{(COMMENT~pltotfm~#2.pl~#2.tfm)}
  \out_line{}
  \out_line{(COMMENT~THIS~FILE~CAN~THEN~BE~DELETED.)}
  \out_line{}
  \out_line{(DESIGNSIZE~R~10.0)}
  \out_line{(DESIGNUNITS~R~10.0)}
  \out_line{}
  \scaled_design_size=10pt
  \let\setglyph=\iffalse
  \let\endsetglyph=\fi
  \let\setkern=\gobble_three
  \let\setrawglyph=\pl_raw_glyph
  \inputmtx{#1}
  \out_line{}
  \out_line{(COMMENT~END~OF~FILE~#2.pl)}
  \close_out{Raw~font}
}}
```

`\pl_raw_glyph`

```
\def\pl_raw_glyph#1#2#3#4#5#6#7#8{
  \a_count=#4
  \afm_convert\a_dimen=#5;
  \afm_convert\b_dimen=#6;
  \afm_convert\c_dimen=#7;
  \afm_convert\d_dimen=#8;
  \out_line{(CHARACTER~\vpl_int\a_count\space(COMMENT~#1)}
  \out_lline{(CHARWD~\vpl_real\a_dimen)}
  \out_lline{(CHARHT~\vpl_real\b_dimen)}
```

```
    \out_lline{(CHARDP~\vpl_real\c_dimen)}
    \out_lline{(CHARIC~\vpl_real\d_dimen)}
    \out_lline{)}
}
```

16 Font transformations

16.1 Transformable metric files

A *transformable metric file* is a metric file which complies with certain restrictions in its syntax. The only metric commands allowed are

```
\setrawglyph{⟨glyph⟩}{⟨font⟩}{⟨size⟩}{⟨slot⟩}{⟨width⟩}{⟨height⟩}
   {⟨depth⟩}{⟨italic⟩}
\setnotglyph{⟨glyph⟩}{⟨font⟩}{⟨size⟩}{⟨slot⟩}{⟨width⟩}{⟨height⟩}
   {⟨depth⟩}{⟨italic⟩}
\setkern{⟨glyph1⟩}{⟨glyph2⟩}{⟨amount⟩}
\setglyph{⟨glyph⟩} ⟨glyph commands⟩ \endsetglyph
```

where ⟨*glyph*⟩, ⟨*glyph1*⟩, ⟨*glyph2*⟩, and ⟨*font*⟩ are strings without any variable references (no `\str` or `\strint` are allowed), ⟨*slot*⟩, ⟨*width*⟩, ⟨*height*⟩, ⟨*depth*⟩, ⟨*italic*⟩, and ⟨*amount*⟩ are TEX numbers, and ⟨*size*⟩ is a TEX dimen. (More accurately, all dimens in a transformable metric file should be on the form ⟨*optional signs*⟩ ⟨*decimal constant*⟩⟨*physical unit*⟩, but that's at the "dangerous bend" level.)

The only ⟨*glyph commands*⟩ allowed are

```
\samesize{⟨glyph⟩}
\glyphpcc{⟨glyph⟩}{⟨xoffset⟩}{⟨yoffset⟩}
```

where ⟨*glyph*⟩ is as above, and ⟨*xoffset*⟩ and ⟨*yoffset*⟩ are TEX numbers.

The only general commands allowed are

```
\setint{⟨name⟩}{⟨number⟩}
\setdim{⟨name⟩}{⟨dimen⟩}
\setstr{⟨name⟩}{⟨string⟩}
```

where ⟨*name*⟩ and ⟨*string*⟩ are strings without variable references, ⟨*number*⟩ is a TEX number, and ⟨*dimen*⟩ is a TEX dimen.

The metric files produced by `\afmtomtx` and `\generalpltomtx` are meant to be transformable. If they are not then there is a bug somewhere.

The name of the integer in `\setint` commands are intepreted. This name is used to determine how the number should be transformed, see the implementation of `\mtxtomtx_setint` below.

16.2 Making font transformations

`\transformfont` The macro:

```
\transformfont{⟨font-name⟩}{⟨transformed font⟩}
```

transforms the metrics of a raw font. As far as TEX is concerned, ⟨*font-name*⟩ will be a new font. It is up to some other piece of software, in most cases the DVI driver, to implement this transformation and provide the transformed font.

`\fromafm` `\frommtx` `\frompl` `\scalefont` `\xscalefont` `\yscalefont` `\slantfont` `\reencodefont` The ⟨*transformed font*⟩ commands are:

`\fromafm{`⟨*AFM file*⟩`}`
`\frompl{`⟨*PL file*⟩`}`
`\fromplgivenetx{`⟨*PL file*⟩`}{`⟨*etx*⟩`}`
`\frommtx{`⟨*MTX file*⟩`}`
`\fromany{`⟨*file*⟩`}`
`\scalefont{`⟨*integer expression*⟩`}{`⟨*transformed font*⟩`}`
`\xscalefont{`⟨*integer expression*⟩`}{`⟨*transformed font*⟩`}`
`\yscalefont{`⟨*integer expression*⟩`}{`⟨*transformed font*⟩`}`
`\slantfont{`⟨*integer expression*⟩`}{`⟨*transformed font*⟩`}`
`\reencodefont{`⟨*etx*⟩`}{`⟨*transformed font*⟩`}`

Each `\transformfont` command generates an `.mtx` file for ⟨*font-name*⟩ and a corresponding raw `.pl` file, which is written out by `\mtxtopl`.

Each `\fromafm`, `\frompl`, or `\fromplgivenetx` command also generates an `.mtx` file for the source font, which is written out by `\afmtomtx` or `\generalpltomtx`. In addition, `\fromafm` also uses `\mtxtopl` to generate a corresponding raw `.pl` file.

`\fromany` reads an MTX, PL, AFM, or VPL file depending on what it can find. It tries them in the order first MTX, then PL, then AFM, and last VPL.

`\transformfont`

```
\def\transformfont#1#2{{
   \unsetstr{afm-name}
   \unsetstr{etx-name}
   \x_resetint{x-scale}{\one_thousand}
   \x_resetint{y-scale}{\one_thousand}
   \x_resetint{slant-scale}{0}
   #2

   \ifisstr{transform-source}\then
      \ifisstr{etx-name}\then
         \edef\a_macro{\string\reencodefont{\str{etx-name}}}
      \else
         \let\a_macro=\empty_command
      \fi
      \_a_true
      \ifnum \int{x-scale}=\one_thousand
         \ifnum \int{slant-scale}=\z@
            \ifnum \int{y-scale}=\one_thousand
               \_a_false
      \fi \fi \fi
      \if_a_
         \edef\a_macro{\a_macro
            \string\transformfont{\strint{x-scale}}{\strint{slant-scale}}
               {\strint{y-scale}}
         }
      \fi
      \record_transform{#1}{\str{transform-source}}{\a_macro}
      \mtxtomtx{\str{afm-name}}{#1}
      \mtxtopl{#1}{#1}
   \else
      \errmessage{Failed~to~transform~font~\str{afm-name};^^J
         font~metrics~file~not~found.
```

```
      }
    \fi
}}
```

\fromafm

```
\def\fromafm#1{
    \x_setstr{afm-name}{#1}
    \afmtomtx{#1}{#1}
    \resetstr{transform-source}{\string\fromafm{#1}{\afm_font_name}}
    \record_transform{#1}{\str{transform-source}}{}
    \mtxtopl{#1}{#1}
}
```

\frommtx

```
\def\frommtx#1{
    \x_setstr{afm-name}{#1}
    \resetstr{transform-source}{\string\frommtx{#1}}
}
```

\frompl
\fromplgivenetx

```
\def\frompl#1{
    \x_setstr{afm-name}{#1}
    \generalpltomtx{#1}{#1}{pl}{}
    \resetstr{transform-source}{\string\frompl{#1}}
    \record_transform{#1}{\string\frompl{#1}}{}
}
```

```
\def\fromplgivenetx#1#2{
    \x_setstr{afm-name}{#1}
    \generalpltomtx{#1}{#1}{pl}{#2}
    \resetstr{transform-source}{\string\frompl{#1}}
    \record_transform{#1}{\string\frompl{#1}}{}
}
```

\fromvpl
\fromvplgivenetx

I realized that there isn't any point in reading metrics for a font that is to be transformed from a VPL file, since no driver I know of can transform virtual fonts. If someone has a problem with this then I suppose he or she should send word about it. /LH

```
% \def\fromvpl#1{
%     \x_setstr{afm-name}{#1}
%     \generalpltomtx{#1}{#1}{vpl}{}
%     \resetstr{transform-source}{\string\frompl{#1}}
% }
```

```
% \def\fromvplgivenetx#1#2{
%     \x_setstr{afm-name}{#1}
%     \generalpltomtx{#1}{#1}{vpl}{#2}
%     \resetstr{transform-source}{\string\frompl{#1}}
% }
```

\fromany

The \fromany transformed font command searches for font metrics for #1 by looking for, in turn, the files #1.mtx, #1.pl, #1.afm, and #1.vpl. If an MTX file doesn't exist, it is generated, and if the MTX is generated from an AFM then a corresponding (non-ligful) PL file is generated as well. \fromany also sets the

fontinst string `transform-source` according to what kind of font it found. If none of the fonts existed then `transform-source` is unset.

```
\def\fromany#1{
   \x_setstr{afm-name}{#1}
   \if_file_exists{#1.mtx}\then
      \resetstr{transform-source}{\string\frommtx{#1}}
   \else
```

1997/01/15 SPQR changed the below search order to `.pl` before `.afm` because of the `cmr*.afm` files found in the `TEXMF/fonts/afm` hierarchy.

```
      \if_file_exists{#1.pl}\then
         \generalpltomtx{#1}{#1}{pl}{}
         \resetstr{transform-source}{\string\frompl{#1}}
         \record_transform{#1}{\string\frompl{#1}}{}
      \else
         \if_file_exists{#1.afm}\then
            \afmtomtx{#1}{#1}
            \resetstr{transform-source}{
               \string\fromafm{#1}{\afm_font_name}
            }
            \record_transform{#1}{\str{transform-source}}{}
            \mtxtopl{#1}{#1}
         \else
            \if_file_exists{#1.vpl}\then
               \generalpltomtx{#1}{#1}{vpl}{}
               \resetstr{transform-source}{\string\fromvpl}
               \record_transform{#1}{\string\fromvpl}{}
            \else
               \unsetstr{transform-source}
            \fi
         \fi
      \fi
   \fi
}
```

The mathematical basis for the metric font transformations

Mathematically, all the metric font transformations (`\scalefont`, `\xscalefont`, `\yscalefont`, and `\slantfont`) are linear mappings of the real plane onto itself. All quantities in a transformable metric file are interpreted as being determined by some point in this plane and hence their transformation depends on how that point would be moved by the metric font transformations performed. This is usually simpler than it sounds, since all quantities except `italicslant` are interpreted as either the x- or the y-coordinate of some point. `italicslant` is interpreted as the quotient $\frac{x}{y}$ for a point.

The best way to describe a linear mapping of the real plane to itself is by a 2×2 matrix whose components are real numbers. Since true real numbers are not available in TEX, integers are used instead, with the convention that they are in units of thousandths. In a concrete form this means that 0 represents 0, 500 represents $\frac{1}{2}$, 1000 represents 1, etc. This works just as for the scaling factors used in `\scale`. It also means that the matrix

$$\begin{pmatrix} 1000 & 0 \\ 0 & 1000 \end{pmatrix}$$

represents the identity mapping (the mapping taking everything to itself).

Thinking of points as column vectors (2×1 matrices) with the x-coordinate in the first component and the y-coordinate in the second, the respective elementary metric font transformations correspond to the following matrices:

$$\texttt{\textbackslash scalefont\{}\langle n\rangle\texttt{\}} \quad \text{is} \quad \begin{pmatrix} \langle n\rangle & 0 \\ 0 & \langle n\rangle \end{pmatrix}$$

$$\texttt{\textbackslash xscalefont\{}\langle n\rangle\texttt{\}} \quad \text{is} \quad \begin{pmatrix} \langle n\rangle & 0 \\ 0 & 1000 \end{pmatrix}$$

$$\texttt{\textbackslash yscalefont\{}\langle n\rangle\texttt{\}} \quad \text{is} \quad \begin{pmatrix} 1000 & 0 \\ 0 & \langle n\rangle \end{pmatrix}$$

$$\texttt{\textbackslash slantfont\{}\langle n\rangle\texttt{\}} \quad \text{is} \quad \begin{pmatrix} 1000 & \langle n\rangle \\ 0 & 1000 \end{pmatrix}$$

Since all these matrices are upper triangular, all products of such matrices (corresponding to compositions of the linear mappings) will be upper triangular as well. It is therefore unnecessary to store the subdiagonal component anywhere (it is always zero), and hence fontinst represents an arbitrary metric transform by the matrix

$$\begin{pmatrix} \texttt{x-scale} & \texttt{slant-scale} \\ 0 & \texttt{y-scale} \end{pmatrix}$$

where `x-scale`, `y-scale`, and `slant-scale` are fontinst integers.

The reason there is a representation of arbitrary metric transforms is that all the elementary metric transforms listed in the second argument of `\transformfont` are concatenated before the actual font file conversion is made. This reduces the amount of calculations performed in case there are many transformations of the font.

Why do we only consider transformations that correspond to upper triangular matrices? Well, a transformation corresponds to an upper triangular matrix if and only if it leaves horizontal lines horizontal. Since in particular the baseline must always be horizontal in TEX, there is no point in considering other linear transformations.

\scalefont
\xscalefont
\yscalefont
\slantfont
\reencodefont

```
\def\scalefont#1#2{
   \eval_expr_to\d_count{#1}
   \x_resetint{x-scale}{\scale{\d_count}{\int{x-scale}}}
   \x_resetint{y-scale}{\scale{\d_count}{\int{y-scale}}}
   \x_resetint{slant-scale}{\scale{\d_count}{\int{slant-scale}}}
   #2
}
\def\xscalefont#1#2{
   \x_resetint{x-scale}{\scale{#1}{\int{x-scale}}}
   #2
}
\def\yscalefont#1#2{
   \eval_expr_to\d_count{#1}
   \x_resetint{y-scale}{\scale{\d_count}{\int{y-scale}}}
   \x_resetint{slant-scale}{\scale{\d_count}{\int{slant-scale}}}
   #2
```

```
}
\def\slantfont#1#2{
   \x_resetint{slant-scale}{
      \add{\scale{#1}{\int{x-scale}}}{\int{slant-scale}}
   }
   #2
}
\def\reencodefont#1#2{
   #2
   \resetstr{etx-name}{#1}
}
```

`\mtxtomtx` The macro:

`\mtxtomtx{`⟨*source MTX*⟩`}{`⟨*destination MTX*⟩`}`

converts the first `.mtx` file to the second, using the current values of `\int{x-scale}`, `\int{y-scale}`, `\int{slant-scale}` and `\str{etx-name}`.

NOTE: this doesn't convert arbitrary `.mtx` files, just the transformable ones.

`\mtxtomtx`

```
\def\mtxtomtx#1#2{{
   \ifisstr{etx-name}\then
      \def\do_slot{\x_resetint\slot_name\slot_number}
      \inputetx{\str{etx-name}}
   \fi
   \open_out{\temp_prefix#2.mtx}
   \def\raw_font_name{#2}
   \out_line{\percent_char~Filename:~#2.mtx}
   \out_line{\percent_char~Created~by:~tex~\jobname}
   \out_line{\percent_char~Created~using:~\string\mtxtomtx{#1}{#2}}
   \out_line{}
   \out_line{\percent_char~This~file~is~used~by~the~fontinst~package.}
   \out_line{}
   \out_line{\percent_char~THIS~FILE~CAN~BE~DELETED.}
   \out_line{}
   \out_line{\string\relax}
   \out_line{\string\metrics}
   \out_line{}
   \out_line{\string\needsfontinstversion{\fontinstversion}}
   \out_line{}
   \let\setint=\mtxtomtx_setint
   \let\setdim=\mtxtomtx_setdim
   \let\setstr=\mtxtomtx_setstr
   \let\setrawglyph=\mtxtomtx_setrawglyph
   \let\setnotglyph=\mtxtomtx_setrawglyph
   \let\setkern=\mtxtomtx_setkern
   \let\setglyph=\mtxtomtx_setglyph
   \let\glyphpcc=\mtxtomtx_glyphpcc
   \let\samesize=\mtxtomtx_samesize
   \let\endsetglyph=\mtxtomtx_endsetglyph
   \inputmtx{#1}
   \out_line{}
   \out_line{\endmetrics_text}
```

```
    \close_out{Transformed~metrics}
}}
```

`\mtxtomtx_setint` `\italicslant_name`

Currently all integers except `italicslant` are interpreted as being the y-coordinate of some point. `italicslant` is interpreted as the quotient $\frac{x}{y}$ for a point (x, y), but represented as a real number (i.e., the TEX number is really a thousand times the actual quotient).

If some new integer is added that cannot be transformed as a y-coordinate, then the `\mtxtomtx_setint` macro must be rewritten.

```
\def\mtxtomtx_setint#1#2{
    \def\a_macro{#1}
    \ifx\a_macro\italicslant_name
       \eval_expr{#2}
       \global\multiply \result \int{x-scale}
       \a_count=\int{slant-scale}
       \multiply \a_count \one_thousand
       \advance \a_count \result
       \divide \a_count \int{y-scale}\x_relax
    \else
       \eval_expr_to\a_count{\scale{#2}{\int{y-scale}}}
    \fi
    \out_line{\string\setint{#1}{\the\a_count}}
}
\def\italicslant_name{italicslant}
```

`\mtxtomtx_setdim` `\mtxtomtx_setstr`

Strings and dimens are not affected by the `\mtxtomtx` transforms.

```
\def\mtxtomtx_setdim#1#2{
    \out_line{\string\setdim{#1}{#2}}
}
```

```
\def\mtxtomtx_setstr#1#2{
    \out_line{\string\setstr{#1}{#2}}
}
```

`\mtxtomtx_setrawglyph`

`#5` (the width) is transformed as the x-coordinate of a point on the baseline. `#6` and `#7` (the height and depth respectively) are transformed as y-coordinates. The depth should probably really have been transformed as the negative of a y-coordinate, but it comes out the same in the end anyway. `#8` (the italic correction) is transformed as the x-coordinate of a point whose y-coordinate equals the height of the character. If the italic slant of the font is negative then it should probably be the depth instead, but it is hard to say for sure how that case should be treated.

```
\def\mtxtomtx_setrawglyph#1#2#3#4#5#6#7#8{
    \eval_expr_to\a_count{\scale{#5}{\int{x-scale}}}
    \eval_expr_to\b_count{\scale{#6}{\int{y-scale}}}
    \eval_expr_to\c_count{\scale{#7}{\int{y-scale}}}
    \eval_expr_to\d_count{#8} \eval_expr{#6}
    \multiply \d_count \int{x-scale}
    \global\multiply \result \int{slant-scale}
    \global\advance \result \d_count
    \rounded_thousandths
    \d_count=\result
    \ifisstr{etx-name}\then
       \ifisint{#1}\then
```

```
673          \e_count=\int{#1}
674       \else
675          \e_count=-1
676       \fi
677    \else
678       \e_count=#4
679    \fi
680    \out_line{
681       \ifnum\e_count>-1
682          \string\setrawglyph
683       \else
684          \string\setnotglyph
685       \fi
686          {#1}{\raw_font_name}{#3}{\the\e_count}
687          {\the\a_count}{\the\b_count}{\the\c_count}{\the\d_count}
688    }
689 }
```

`\mtxtomtx_setkern` Kerns are transformed as the x-coordinate of a point on the baseline.

```
690 \def\mtxtomtx_setkern#1#2#3{
691    \eval_expr{\scale{\int{x-scale}}{#3}}
692    \out_line{\string\setkern{#1}{#2}{\the\result}}
693 }
```

`\mtxtomtx_setglyph`
`\mtxtomtx_samesize`
`\mtxtomtx_endsetglyph`

```
694 \def\mtxtomtx_setglyph#1{\out_line{\string\setglyph{#1}}}
695 \def\mtxtomtx_samesize#1{\out_lline{\string\samesize{#1}}}
696 \def\mtxtomtx_endsetglyph{\out_line{\string\endsetglyph}}
```

`\mtxtomtx_glyphpcc` `#2` is transformed as the x-coordinate and `#3` is transformed as the y-coordinate of a point—the same point for both parameters.

```
697 \def\mtxtomtx_glyphpcc#1#2#3{
698    \eval_expr_to\b_count{\scale{\int{y-scale}}{#3}}
699    \eval_expr_to\a_count{#2} \eval_expr{#3}
700    \multiply \a_count \int{x-scale}
701    \global\multiply \result \int{slant-scale}
702    \global\advance \result \a_count
703    \rounded_thousandths
704    \out_lline{\string\glyphpcc{#1}{\the\result}{\the\b_count}}
705 }
```

16.3 Changing glyph names

fontinst uses names to identify glyphs, and if for example the font in question is a postscript font, then names will also be used to identify glyphs in the printer. Between those two points however, and in particular inside TEX itself, glyphs are represented with numbers (slots). Therefore there is no real need for the glyph names used within fontinst and the glyph names used in the printer (the names gotten from the AFM file) to be equal, but they usually are. There are some cases though where the glyph names of a font are unsuitable for use with fontinst—mainly because fontinst can mix glyphs from different printer fonts—and therefore fontinst also offers the ability to automatically change the names of glyphs in transformable metric files.

\reglyphfonts
\endreglyphfonts

\reglyphfonts ⟨*reglyphing commands*⟩ \endreglyphfonts

```
\def\reglyphfonts{
  \begingroup
  \x_setint{renameweight}{1}
  \x_setint{killweight}{-10}
  \setcommand\iftokeep##1\then{\ifnum -1<##1}
}
\def\endreglyphfonts{\endgroup}
```

The ⟨*reglyphing commands*⟩ are

\renameglyph{⟨*to*⟩}{⟨*from*⟩}
\renameglyphweighted{⟨*to*⟩}{⟨*from*⟩}{⟨*weight*⟩}
\killglyph{⟨*glyph*⟩}
\killglyphweighted{⟨*glyph*⟩}{⟨*weight*⟩}
\offmtxcommand{⟨*command*⟩}
\onmtxcommand{⟨*command*⟩}
\reglyphfont{⟨*destination font*⟩}{⟨*source font*⟩}

The only reglyphing command that actually convert the names of any glyphs is \reglyphfont; it reads a font ⟨*source font*⟩ (which may be of type MTX, PL, AFM, or VPL) and writes another font ⟨*destination font*⟩ in which the names of glyphs have been converted. All the other commands control *how* this conversion should be made, and these settings get cleared at the closing \endreglyphfonts.

\reglyphfont

The conversion works in two ways. First of all, the names of the glyphs can be changed. This works as a general mapping and is controlled by the \renameglyph and \renameglyphweighted commands. Any mensioning of the glyph ⟨*from*⟩ in a command will be converted to a mensioning of the glyph ⟨*to*⟩, if that command survives the conversion. The other way the conversion works is that it can selectively kill—refrain from including in ⟨*destination font*⟩—commands in the metric file. This part weighs in several factors.

\renameglyph
\renameglyphweighted

For one thing, one can specify that all metric commands of a certain type should be killed, and this is done with the \offmtxcommand command. For example, one can see to that all kerning commands are killed by

\offmtxcommand{\setkern}

\offmtxcommand

The effect is the same as that of saying

offkern,⟨*destination font*⟩,onkern

rather than just ⟨*destination font*⟩ in the second argument to \installfont, but it is somewhat faster since less text is written to and subsequently read from the ⟨*destination font*⟩.mtx file. The effect of a previous \offmtxcommand can be canceled by a call to \onmtxcommand, just like with \offcommand and \oncommand.

\onmtxcommand

The survivance of a command is also affected by the glyphs it refers to. Each glyph has a *weight* associated with it and the sum of the weights for all glyphs mensioned by a command is also used to decide whether that command should survive. The test here is performed by the macro \iftokeep, whose parameter text must be #1\then, where #1 will be a \count register. This macro must eventually expand to an if of some sort and that if evaluating to true is interpreted as that the command should be kept. The default replacement text is \ifnum -1<#1, which causes a command to be killed (not kept) iff the sum of weights for it is negative.

\iftokeep

The weight of a glyph is set by `\renameglyph`, `\renameglyphweighted`, `\killglyph`, and `\killglyphweighted`. The `\rename...` commands also set a new name for the glyph if it survives, whereas the `\kill...` commands will keep the old name. Since the standard settings are that a `\rename...` weight is small and positive and a `\kill...` weight is large and negative, glyphs for which a `\kill...` has been done will usually not survive.

`\killglyph` `\killglyphweighted`

Any one of `\renameglyph`, `\renameglyphweighted`, `\killglyph`, and `\killglyphweighted` for a glyph will override all previous settings by any of these four commands for that glyph. The equivalent of the neutral state for a glyph (no settings by any of these commands have been made for that glyph) is achieved by the command

```
\killglyphweighted{⟨glyph⟩}{0}
```

These two are just special forms of `\offcommand` and `\oncommand`.

`\offmtxcommand` `\onmtxcommand`

```
\def\offmtxcommand#1{
   \x_cs\offcommand{reglyph_\expandafter\gobble_one\string#1}
}
\def\onmtxcommand#1{
   \x_cs\oncommand{reglyph_\expandafter\gobble_one\string#1}
}
```

Inside a `\reglyphfonts` ... `\endreglyphfonts` block, the family `\slots-`⟨*glyph*⟩ of control sequences is used to store the information about how glyph ⟨*glyph*⟩ should be converted. These control sequences are either undefined or parameterless macros whose replacement texts are of one of the forms

`\slots-GLYPH`

```
\rename_glyph{⟨to⟩}{⟨weight⟩}
\rename_glyph{⟨to⟩}\i-renameweight
\kill_glyph{⟨weight⟩}
\kill_glyph\i-killweight
```

⟨*to*⟩ is what the glyph will be renamed to and ⟨*weight*⟩ is the associated weight. These four different forms are generated by the four different commands `\renameglyphweighted`, `\renameglyph`, `\killglyphweighted`, and `\killglyph` respectively.

The difference between the commands `\renameglyphweighted` and `\renameglyph` is that the former lets one specify the weight exactly while the latter will use the value of the integer `renameweight` *at the time of conversion*. By changing the value of `renameweight` between two conversions, one changes the weights used for all glyph renamings declared using `\renameglyph`.

`\renameglyph` `\renameglyphweighted`

```
\begingroup
   \catcode`\-=11
   \gdef\renameglyph#1#2{
      \x_cs\edef{slots-#2}{
         \noexpand\rename_glyph{#1}\noexpand\i-renameweight
      }
   }
\endgroup
\def\renameglyphweighted#1#2#3{
   \eval_expr{#3}
   \x_cs\edef{slots-#2}{\noexpand\rename_glyph{#1}{\the\result}}
}
```

\killglyph
\killglyphweighted

The difference between the commands `\killglyphweighted` and `\killglyph` is that the former lets one specify the weight exactly while the latter will use the value of the integer `killweight` *at the time of conversion*. By changing the value of `killweight` between two conversions, one changes the weights used for all glyph killings declared using `\killglyph`.

```
\begingroup
   \catcode`\-=11
   \gdef\killglyph#1{
      \x_cs\def{slots-#1}{\kill_glyph\i-killweight}
   }
\endgroup
\def\killglyphweighted#1#2{
   \eval_expr{#2}
   \x_cs\edef{slots-#1}{\noexpand\kill_glyph{\the\result}}
}
```

\reglyphfont

The command

`\reglyphfont{`⟨*destination font*⟩`}{`⟨*source font*⟩`}`

reads the font metric file ⟨*source font*⟩`.mtx` (which must be transformable), ⟨*source font*⟩`.pl`, or ⟨*source font*⟩`.afm` (the possibilities are tried in that order) and writes out a font metric file ⟨*destination font*⟩`.mtx` that is the converted (as described above) form of the source font.

⟨*destination font*⟩ and ⟨*source font*⟩ may not be equal.

```
\def\reglyphfont#1#2{
   \fromany{#2}
   \ifisstr{transform-source}\then
      \reglyph_font{#1}{#2}
      \record_transform{#1}{\str{transform-source}}{\string\reglyphfont}
   \else
      \errmessage{Could~not~find~font~metrics~for~#2.}
   \fi
}
```

\reglyph_font

This macro does the actual conversion.

```
\def\reglyph_font#1#2{{
   \open_out{\temp_prefix#1.mtx}
   \out_line{\percent_char~Filename:~#1.mtx}
   \out_line{\percent_char~Created~by:~tex~\jobname}
   \out_line{\percent_char~Created~using:~\string\reglyphfont{#1}{#2}}
   \out_line{}
   \out_line{\percent_char~This~file~is~used~by~the~fontinst~package.}
   \out_line{}
   \out_line{\percent_char~THIS~FILE~CAN~BE~DELETED.}
   \out_line{}
   \out_line{\string\relax}
   \out_line{\string\metrics}
   \out_line{}
   \out_line{\string\needsfontinstversion{\fontinstversion}}
   \out_line{}
   \let\setint=\reglyph_setint
   \let\setdim=\reglyph_setdim
```

```
    \let\setstr=\reglyph_setstr
    \let\setrawglyph=\reglyph_setrawglyph
    \let\setnotglyph=\reglyph_setrawglyph
    \let\setkern=\reglyph_setkern
    \let\setglyph=\reglyph_setglyph
    \let\glyphpcc=\reglyph_glyphpcc
    \let\samesize=\reglyph_samesize
    \let\endsetglyph=\reglyph_endsetglyph
    \inputmtx{#2}
    \out_line{}
    \out_line{\endmetrics_text}
    \close_out{Reglyphed~metrics}
}}
```

\reglyph_setint
\reglyph_setdim
\reglyph_setstr

These are just copied to the file generated.

```
\def\reglyph_setint#1#2{\out_line{\string\setint{#1}{#2}}}
\def\reglyph_setdim#1#2{\out_line{\string\setdim{#1}{#2}}}
\def\reglyph_setstr#1#2{\out_line{\string\setstr{#1}{#2}}}
```

\command_survivance

This `\count` register stores the sum of the weights associated with the glyphs considered so far. It is updated by `\rename_glyph` and `\kill_glyph`.

```
\newcount\command_survivance
```

\glyphname

The macro `\glyphname` holds the name of the glyph currently under consideration. It is altered by `\rename_glyph`.

\rename_glyph
\kill_glyph

```
\def\rename_glyph#1{\def\glyphname{#1}\kill_glyph}
\def\kill_glyph#1{\advance \command_survivance #1\x_relax}
```

\reglyph_setrawglyph

The `\reglyph_setrawglyph` macro is straightforward. Whether it writes a `\setrawglyph` or a `\setnotglyph` command depends on the fourth parameter, just like with `\mtxtomtx_setrawglyph`.

```
\def\reglyph_setrawglyph#1#2#3#4#5#6#7#8{
    \command_survivance=0
    \def\glyphname{#1}
    \csname slots-#1\endcsname
    \iftokeep\command_survivance\then
       \out_line{
          \ifnum #4<\z@
             \string\setnotglyph
          \else
             \string\setrawglyph
          \fi
          {\glyphname}{#2}{#3}{#4}{#5}{#6}{#7}{#8}
       }
    \fi
}
```

\reglyph_setkern

```
\def\reglyph_setkern#1#2#3{
    \command_survivance=0
    \def\glyphname{#1}
```

```
   \csname slots-#1\endcsname
   \let\a_macro=\glyphname
   \def\glyphname{#2}
   \csname slots-#2\endcsname
   \iftokeep\command_survivance\then
      \out_line{\string\setkern{\a_macro}{\glyphname}{#3}}
   \fi
}
```

\reglyph_setglyph
\off-\reglyph_setglyph
\reglyph_glyphpcc
\reglyph_samesize
\reglyph_endsetglyph

In \setglyph ... \endsetglyph constructions (which are written for composite characters in AFM files), the decision of whether to write a command or not due to glyph weights is done only once for the entire construction. This means that the commands must be saved until the \endsetglyph where the result is finally known. The token list register \a_toks is used for this purpose.

```
\def\reglyph_setglyph#1{
   \command_survivance=0
   \def\glyphname{#1}
   \csname slots-#1\endcsname
   \edef\a_macro{\string\out_line{\string\setglyph{\glyphname}}}
   \a_toks=\expandafter{\a_macro}
}
\x_cs\def{off-\string\reglyph_setglyph}#1{\gobble_glyph}

\def\reglyph_glyphpcc#1#2#3{
   \def\glyphname{#1}
   \csname slots-#1\endcsname
   \edef\a_macro{\string\out_lline{
      \string\glyphppc{\glyphname}{#2}{#3}
   }}
   \a_toks=\expandafter{\the\expandafter\a_toks \a_macro}
}

\def\reglyph_samesize#1{
   \def\glyphname{#1}
   \csname slots-#1\endcsname
   \edef\a_macro{\string\out_lline{
      \string\samesize{\glyphname}
   }}
   \a_toks=\expandafter{\the\expandafter\a_toks \a_macro}
}

\def\reglyph_endsetglyph{
   \iftokeep\command_survivance\then
      \the\a_toks
      \out_line{\string\endsetglyph}
   \fi
   \a_toks={}
}
⟨/pkg⟩
```

Two common reglyphing schemes

As is mensioned elsewhere, the most common reglyphing operation is to take a caps and small caps font produced by some major foundry and change the glyph names so that they agree with the glyph names used in expert fonts. The following

code contains the modifying reglyphing commands to set up this reglyphing, in two different variants.

The commands are currently based on a comparision of Adobe Garamond Small Caps & Oldstyle Figures (`padrc8a` in the Berry scheme) with Adobe Garamond Regular Expert (`padr8x` in the Berry scheme), so they should be correct for a fair amount of Adobe font families, but it is also highly probable that there are lots of fonts out there for which it doesn't work quite right. In case you do find such a font, please write to tell the fontinst mailing list about it—it would be rather easy to add various alternative setup schemes, controlled by switches, to these files. Just make sure first (by checking the newest version of fontinst) that the alternative setting you have found hasn't already been included.

As mensioned, there are two different reglyphing schemes that are set up by the code below—one has docstrip guard ⟨glyphs⟩, the other has guard ⟨!glyphs⟩—but they both change CSC names to Expert names. The difference lies instead in what information is copied from source font to destination font: the ⟨glyphs⟩ variant copies everything, whereas the ⟨!glyphs⟩ variant doesn't copy `\setrawglyph` commands, `\setnotglyph` commands, or `\setglyph` constructions. The ⟨!glyphs⟩ variant also suppresses kerns between two glyphs that doesn't change name.

The motive for having such a curious setup naturally lies in how the files are meant to be used. If you have CSC fonts, but no Expert fonts, then you should definately use the ⟨glyphs⟩ variant. If on the other hand you have both CSC and Expert fonts for a family, then it is worth considering using the ⟨!glyphs⟩ variant instead. The observation this is based on is that within a triad of the corresponding regular, expert, and CSC fonts, almost all glyphs present in the CSC font can also be found in either the regular or the expert font; furthermore the only missing glyphs were `FIsmall`, `FLsmall`, and `SSsmall`, which (i) were included in the CSC font only to complete the `8a` encoding vector and (ii) are identical to fontinst's fakes for them. Thus by constructing the `sc` shape fonts from the regular and expert variants, instead of the CSC variant, one can get away with using one raw font less, thus reducing the time needed for downloading the fonts to the printer. One thing not found in either of the regular or expert font in the triad is however the kerns between capitals and small capitals, but these can be extracted from the metrics of the CSC font, and doing this is the primary objective for the ⟨!glyphs⟩ variant.

First there is the English alphabet:

```
⟨*reglyph⟩
\renameglyph{Asmall}{a}
\renameglyph{Bsmall}{b}
\renameglyph{Csmall}{c}
\renameglyph{Dsmall}{d}
\renameglyph{Esmall}{e}
\renameglyph{Fsmall}{f}
\renameglyph{Gsmall}{g}
\renameglyph{Hsmall}{h}
\renameglyph{Ismall}{i}
\renameglyph{Jsmall}{j}
\renameglyph{Ksmall}{k}
\renameglyph{Lsmall}{l}
\renameglyph{Msmall}{m}
\renameglyph{Nsmall}{n}
```

```
\renameglyph{Osmall}{o}
\renameglyph{Psmall}{p}
\renameglyph{Qsmall}{q}
\renameglyph{Rsmall}{r}
\renameglyph{Ssmall}{s}
\renameglyph{Tsmall}{t}
\renameglyph{Usmall}{u}
\renameglyph{Vsmall}{v}
\renameglyph{Wsmall}{w}
\renameglyph{Xsmall}{x}
\renameglyph{Ysmall}{y}
\renameglyph{Zsmall}{z}
```

Then there are the figures:

```
\renameglyph{zerooldstyle}{zero}
\renameglyph{oneoldstyle}{one}
\renameglyph{twooldstyle}{two}
\renameglyph{threeoldstyle}{three}
\renameglyph{fouroldstyle}{four}
\renameglyph{fiveoldstyle}{five}
\renameglyph{sixoldstyle}{six}
\renameglyph{sevenoldstyle}{seven}
\renameglyph{eigtholdstyle}{eight}
\renameglyph{nineoldstyle}{nine}
```

Then there are the accents and a couple of miscellaneous symbols. You might want to check these carefully, as there might not always be a distinction.

```
\renameglyph{Gravesmall}{grave}
\renameglyph{Acutesmall}{acute}
\renameglyph{Circumflexsmall}{circumflex}
\renameglyph{Tildesmall}{tilde}
\renameglyph{Macronsmall}{macron}
\renameglyph{Brevesmall}{breve}
\renameglyph{Dotaccentsmall}{dotaccent}
\renameglyph{Dieresissmall}{dieresis}
\renameglyph{Ringsmall}{ring}
\renameglyph{Cedillasmall}{cedilla}
\renameglyph{Ogoneksmall}{ogonek}
\renameglyph{Caronsmall}{caron}
\renameglyph{dollaroldstyle}{dollar}
\renameglyph{ampersandsmall}{ampersand}
\renameglyph{centoldstyle}{cent}
```

There are also all the non-English letters:

```
\renameglyph{AEsmall}{ae}
\renameglyph{Ethsmall}{eth}
\renameglyph{Lslashsmall}{lslash}
\renameglyph{Oslashsmall}{oslash}
\renameglyph{OEsmall}{oe}
\renameglyph{Thornsmall}{thorn}

\renameglyph{Agravesmall}{agrave}
\renameglyph{Egravesmall}{egrave}
\renameglyph{Igravesmall}{igrave}
\renameglyph{Ogravesmall}{ograve}
\renameglyph{Ugravesmall}{ugrave}
```

```
\renameglyph{Aacutesmall}{aacute}
\renameglyph{Eacutesmall}{eacute}
\renameglyph{Iacutesmall}{iacute}
\renameglyph{Oacutesmall}{oacute}
\renameglyph{Uacutesmall}{uacute}
\renameglyph{Yacutesmall}{yacute}

\renameglyph{Acircumflexsmall}{acircumflex}
\renameglyph{Ecircumflexsmall}{ecircumflex}
\renameglyph{Icircumflexsmall}{icircumflex}
\renameglyph{Ocircumflexsmall}{ocircumflex}
\renameglyph{Ucircumflexsmall}{ucircumflex}

\renameglyph{Atildesmall}{atilde}
\renameglyph{Ntildesmall}{ntilde}
\renameglyph{Otildesmall}{otilde}

\renameglyph{Adieresissmall}{adieresis}
\renameglyph{Edieresissmall}{edieresis}
\renameglyph{Idieresissmall}{idieresis}
\renameglyph{Odieresissmall}{odieresis}
\renameglyph{Ydieresissmall}{ydieresis}
\renameglyph{Udieresissmall}{udieresis}

\renameglyph{Aringsmall}{aring}
\renameglyph{Ccedillasmall}{ccedilla}
\renameglyph{Scaronsmall}{scaron}
\renameglyph{Zcaronsmall}{zcaron}
```

The following four glyphs aren't really necessary, since they are usually identical to another glyph or to their fakes.

```
\killglyph{dotlessi}
⟨*glyphs⟩
\renameglyph{FIsmall}{fi}
\renameglyph{FLsmall}{fl}
\renameglyph{SSsmall}{germandbls}
⟨/glyphs⟩
⟨*!glyphs⟩
\killglyph{fi}
\killglyph{fl}
\killglyph{germandbls}
⟨/!glyphs⟩
```

The ⟨!glyphs⟩ settings have no need for `\setrawglyph` or `\setglyph` commands, and no need for the kerns between capitals either, since these are already known from the regular variant.

```
⟨*!glyphs⟩
\offmtxcommand\setrawglyph
\offmtxcommand\setglyph
\resetcommand\iftokeep#1\then{\ifnum 0<#1}
⟨/!glyphs⟩
⟨/reglyph⟩
```

File d
filtfam.dtx

17 Installing Latin families

\latinfamily The macro:

\latinfamily{⟨*family*⟩}{⟨*commands*⟩}

installs a Latin font family.

```
⟨*pkg⟩
\def\latinfamily#1#2{{
   \edef\a_macro{#1}
   \expandafter\parse_family\a_macro
      \empty_command\empty_command\end_parse_family
   \installfonts
      \if_file_exists{\raw_encoding.mtx}\then
         \installfamily{\raw_encoding}{\latex_family}{#2}
      \fi
      \installfamily{OT1}{\latex_family}{#2}
      \installfamily{T1}{\latex_family}{#2}
⟨*textcomp⟩
      \installfamily{TS1}{\latex_family}{#2}
⟨/textcomp⟩
      \latin_weights
   \endinstallfonts
}}
```

\parse_family \font_family \font_variant \raw_variant \latex_variant

\parse_family FAMILY\end_parse_family

Intializes \latex_family to FAMILY, \font_family to first three characters of FAMILY, \font_variant and \raw_variant to fourth and fifth character (if present).

SPQR 02/95: If the fourth parameter is x or 9, then its an expert indication, not a variant. So initialy set \raw_variant to #4, but unset it if the expert test succeeds. When #4 is used for a real variant, it passes through.

SPQR 05/95: If it *is* an expert font, then the variantness is expressed by the encoding, not a variant letter, so unset \font_variant.

UV 06/98: Added \if_oldstyle_ code from Alan's v 1.511 beta. If the fourth parameter is 9, treat it as if j was given.

Note that instead of using \setcommand\digit, we now use separate encoding files for T1j.etx, etc.

```
\def\parse_family #1#2#3#4#5\end_parse_family{
   \gdef\font_family{#1#2#3}
   \gdef\font_variant{#4#5}
   \gdef\raw_variant{#4#5}
   \gdef\latex_family{#1#2#3#4#5}
   \_expert_false
   \_oldstyle_false
   \ifx#4x
      \_expert_true
      \gdef\raw_variant{#5}
```

```
      \gdef\font_variant{#5}
      %\setcommand\digit##1{##1}
   \else\ifx#4j
      \_expert_true
      \_oldstyle_true
      \gdef\raw_variant{#5}
      \gdef\font_variant{#5}
      %\setcommand\digit##1{##1oldstyle}
   \else\ifx#49
      \gdef\latex_family{#1#2#3j#5}
      \_expert_true
      \_oldstyle_true
      \gdef\raw_variant{#5}
      \gdef\font_variant{#5}
      %\setcommand\digit##1{##1oldstyle}
   \else
      \_expert_false
      \_oldstyle_false
      %\setcommand\digit##1{##1}
   \fi\fi\fi
⟨*debug⟩
   \if_oldstyle_
   \immediate\write16{INFO>~parse~family~<#1#2#3><#5>~(oldstyle)}
   \else\if_expert_
   \immediate\write16{INFO>~parse~family~<#1#2#3><#5>~(expert)}
   \else
   \immediate\write16{INFO>~parse~family~<#1#2#3><#4#5>}
   \fi\fi
⟨/debug⟩
}
```

\if_expert_
\if_oldstyle_

```
\newif\if_expert_
\newif\if_oldstyle_
```

17.1 Processing a list of weights, widths and shapes

\latin_weight
\font_weight
\latex_weight

\latin_weight{FONTWEIGHT}{LATEXWEIGHT}

```
\def\latin_weight#1#2{
   \gdef\font_weight{#1}%
   \gdef\latex_weight{#2}%
   \latin_widths
}
```

\latin_width
\font_width
\latex_width

\latin_width{FONTWIDTH}{LATEXWIDTH}

```
\def\latin_width#1#2{
   \gdef\font_width{#1}
   \gdef\latex_width{#2}
   \latin_shapes
}
```

\latin_shape
\font_shape
\raw_shape
\encoding_shape
\latex_shape
\encoding_switches

\latin_shape{FONTSHAPE}{RAWSHAPE}{ENCODINGSHAPE}{LATEXSHAPE}{SWITCHES}

Calls `\fake_width_WIDTH` and `\fake_shape_SHAPE` to generate an `8r`-reencoded `.mtx` file. If successful, calls `\latin_encodings`.

UV 06/98: Added a fifth argument `SWITCHES`, which may be used to set `\if_textcomp_` before it is evaluated in `\latin_encodings`.

```
\def\latin_shape#1#2#3#4#5{
   \gdef\font_shape{#1}
   \gdef\raw_shape{#2}
   \gdef\encoding_shape{#3}
   \if_oldstyle_
      \gdef\encoding_shape{#3\oldstyle_shape}
   \fi
   \gdef\latex_shape{#4}
   \gdef\encoding_switches{#5}
⟨*debug⟩
   \immediate\write16{
      INFO>~to~make~LaTeX~font~shape~<\latex_family,
      \latex_weight,\latex_shape,\latex_width>~seek~
      \font_family\font_weight\raw_shape
         \raw_variant\raw_encoding\font_width.mtx}
⟨/debug⟩
   \csname fake_width_\font_width\endcsname
   \csname fake_shape_\raw_shape\endcsname
   \if_file_exists{
      \font_family\font_weight\raw_shape
         \raw_variant\raw_encoding\font_width.mtx
   }\then
      \latin_encodings
   \fi
}
```

17.2 Faking font shapes

(CK) changed font faking code; based on SPQR's code (still in this file). I am not sure that know what I am doing here; let's hope I won't be caught. Also changed: Order of `\font_width` and `\font_encoding` in MANY places. Also added tests which prevent multiple calls to the same font faking routine. (Seems to be necessary in some cases and can't hurt.) — (ASAJ) I think CK knows more about what he's doing than he's letting on.

`\fake_shape_` Check if an `8a`-encoded `.afm` file exists for current combination of font attributes, and, if so, call `\fake_shape_raw_encoding`. For expert fonts, also check if an `8x`-encoded `.afm` file exists, and call `\fake_shape_expert_encoding`, if approporiate.

UV, 06/98: Cut down lengthy `\fake_shape_` routine into several subroutines `\fake_shape_raw_encoding` and optionally `\fake_shape_expert_encoding` and `\fake_shape_oldstyle_encoding`.

UV, 06/98: Added experimental code to install OsF or SC&OsF fonts.

```
\def\fake_shape_{
   \if_file_exists{
      \font_family\font_weight\raw_shape\raw_variant
         \adobe_encoding\font_width.afm
   }\then
      \fake_shape_raw_encoding
```

```
   \fi
   \if_expert_
      \if_file_exists{
         \font_family\font_weight\raw_shape\raw_variant
            \expert_encoding\font_width.afm
      }\then
         \fake_shape_expert_encoding
      \fi
   \fi
⟨*oldstyle⟩
   \if_oldstyle_
      \gdef\oldstyle_variant{\oldstyle_shape}
      \if_file_exists{
         \font_family\font_weight\raw_shape\oldstyle_variant
            \adobe_encoding\font_width.afm
      }\then
         \fake_shape_oldstyle_encoding
      \else
         \gdef\oldstyle_variant{\caps_shape}
         \if_file_exists{
            \font_family\font_weight\raw_shape\oldstyle_variant
               \adobe_encoding\font_width.afm
         }\then
            \fake_shape_oldstyle_encoding
         \fi
      \fi
   \fi
⟨/oldstyle⟩
}
```

`\fake_shape_raw_encoding` Called from `\fake_shape_` if an 8a-encoded `.afm` file was found. Invokes

```
\transformfont{<FONT>8r}
  {\reencodefont{8r}{\fromafm{<FONT>8a}}}
```

to generate an 8r-reencoded `.mtx` file (and a raw `.pl` file), followed by

```
\installrawfont{<FONT>8r}{<FONT>8r,8r}{8r}...
```

to install a ligfull `.pl` file.

```
\def\fake_shape_raw_encoding{
   \if_file_exists{
      \font_family\font_weight\raw_shape\raw_variant
         \raw_encoding\font_width.mtx
   }\then % no action required
   \else
⟨*debug⟩
      \immediate\write16{
         INFO>~run~\string\transformfont\space\space
         <\font_family\font_weight\raw_shape\raw_variant
            \raw_encoding\font_width>~from~
         <\font_family\font_weight\raw_shape\raw_variant
            \adobe_encoding\font_width>
      }
⟨/debug⟩
```

```
      \transformfont{
         \font_family\font_weight\raw_shape\raw_variant
            \raw_encoding\font_width
      }{
         \reencodefont{\raw_encoding}{
            \fromafm{
               \font_family\font_weight\raw_shape\raw_variant
                  \adobe_encoding\font_width
            }
         }
      }
      \if_file_exists{ \raw_encoding.mtx }\then
⟨*debug⟩
         \immediate\write16{
            INFO>~run~\string\installrawfont\space
            <\font_family\font_weight\font_shape\raw_variant
               \raw_encoding\font_width>
            <\font_family\font_weight\raw_shape\raw_variant
               \raw_encoding\font_width,
             \raw_encoding>
            <\raw_encoding>
            <\raw_encoding>
            <\latex_family>
            <\latex_weight\latex_width>
            <\latex_shape>}
⟨/debug⟩
         \installrawfont{
            \font_family\font_weight\raw_shape\raw_variant
               \raw_encoding\font_width
         }{ \font_family\font_weight\raw_shape\raw_variant
               \raw_encoding\font_width,
            \raw_encoding
         }{ \raw_encoding
         }{ \raw_encoding
         }{ \latex_family
         }{ \latex_weight\latex_width
         }{ \latex_shape
         }{}
      \fi
   \fi
}
```

`\fake_shape_expert_encoding` Called from `\fake_shape_` if an 8x-encoded `.afm` file was found. Invokes

```
\fromafm{<FONT>8x}
```

to generate an 8x-encoded `.mtx` file (and a raw `.pl` file).

```
\def\fake_shape_expert_encoding{
   \if_file_exists{
      \font_family\font_weight\raw_shape\raw_variant
         \expert_encoding\font_width.mtx
   }\then % no action required
   \else
      \fromafm{
```

```
          \font_family\font_weight\raw_shape\raw_variant
             \expert_encoding\font_width
       }
    \fi
}
```

`\fake_shape_oldstyle_encoding` Called from `\fake_shape_` if an OsF or SC&OsF variant of an `8a`-encoded `.afm` file was found. Invokes

```
\transformfont{<FONT>j8r}
  {\reencodefont{8r}{\fromafm{<FONT>j8a}}}
```

to generate an `8r`-reencoded `.mtx` file (and a raw `.pl` file).

```
⟨*oldstyle⟩
\def\fake_shape_oldstyle_encoding{
   \if_file_exists{
      \font_family\font_weight\raw_shape\oldstyle_variant
         \adobe_encoding\font_width.mtx
   }\then % no action required
   \else
⟨*debug⟩
      \immediate\write16{
         INFO>~run~\string\transformfont\space\space
         <\font_family\font_weight\raw_shape\oldstyle_variant
            \raw_encoding\font_width>~from~
         <\font_family\font_weight\raw_shape\oldstyle_variant
            \adobe_encoding\font_width>
      }
⟨/debug⟩
      \transformfont{
         \font_family\font_weight\raw_shape\oldstyle_variant
            \raw_encoding\font_width
      }{
         \reencodefont{\raw_encoding}{
            \fromafm{
               \font_family\font_weight\raw_shape\oldstyle_variant
                  \adobe_encoding\font_width
            }
         }
      }
   \fi
}
⟨/oldstyle⟩
```

`\fake_shape_c` Check if an `8a`-encoded `.afm` file exists for the small caps shape. If so, call `\fake_shape_` to generate an `8r`-reencoded MTX file and to install a ligfull `.pl` file. If not, reset `\raw_shape` to the default shape and `\encoding_shape` to small caps before calling `\fake_shape_` to install a faked small caps font.

```
\def\fake_shape_c{
   \if_file_exists{
      \font_family\font_weight\raw_shape\raw_variant
         \adobe_encoding\font_width.afm
   }\then
      \fake_shape_
```

```
234    \else
235       % If real smallcaps font doesn't exist, fake it from the roman.
236       \gdef\raw_shape{}
237       \if_oldstyle_
238          \gdef\encoding_shape{cj}
239       \else
240          \gdef\encoding_shape{c}
241       \fi
242       \fake_shape_
243    \fi
244 }
```

`\fake_shape_o` Check if an `8a`-encoded `.afm` file exists for the oblique shape. If so, call `\fake_shape_` to generate an `8r`-reencoded MTX file and to install a ligfull `.pl` file. If not, call macros to fake a oblique fonts.

```
245 \def\fake_shape_o{
246    \if_file_exists{
247       \font_family\font_weight\raw_shape\raw_variant
248          \adobe_encoding\font_width.afm
249    }\then
250       \fake_shape_
251    \else
252       \fake_shape_o_raw_encoding
253      \if_expert_
254       \fake_shape_o_expert_encoding
255      \fi
256 ⟨*oldstyle⟩
257      \if_oldstyle_
258       \fake_shape_o_oldstyle_encoding
259      \fi
260 ⟨/oldstyle⟩
261    \fi
262 }
```

`\fake_shape_o_raw_encoding` Called from `\fake_shape_o` if the oblique shape needs to be faked. Invokes

```
\transformfont{<FONT>o8r}
  {\slantfont{SLANT}{\frommtx{<FONT>8r}}}
```

to generate an `8r`-reencoded `.mtx` file (and a raw `.pl` file), followed by

```
\installrawfont{<FONT>o8r}{<FONT>o8r,8r}{8r}...
```

to install a ligfull `.pl` file.

```
263 \def\fake_shape_o_raw_encoding{
264    \if_file_exists{
265       \font_family\font_weight\raw_variant
266          \raw_encoding\font_width.mtx
267    }\then
268       \if_file_exists{
269          \font_family\font_weight\font_shape\raw_variant
270             \raw_encoding\font_width.mtx
271       }\then % no action required
272       \else
```

```
⟨*debug⟩
          \immediate\write16{
             INFO>~run~\string\transformfont\space\space
             <\font_family\font_weight\font_shape\raw_variant
                \raw_encoding\font_width>~from~
             <\font_family\font_weight\raw_variant
                \raw_encoding\font_width>~(faking~oblique)
          }
⟨/debug⟩
          %%% WARNING: famtool.pl relies on this message format!!!
          \immediate\write16{
             Faking~oblique~font~
             \font_family\font_weight\font_shape\raw_variant
                \raw_encoding\font_width
             \space from~
             \font_family\font_weight\raw_variant
                \raw_encoding\font_width
          }
          \transformfont{
             \font_family\font_weight\font_shape\raw_variant
                \raw_encoding\font_width
          }{
             \slantfont{\SlantAmount}{
                \frommtx{
                   \font_family\font_weight\raw_variant
                      \raw_encoding\font_width
                }
             }
          }
          \if_file_exists{ \raw_encoding.mtx }\then
⟨*debug⟩
             \immediate\write16{
                INFO>~run~\string\installrawfont\space
                <\font_family\font_weight\font_shape\raw_variant
                   \raw_encoding\font_width>
                <\font_family\font_weight\font_shape\raw_variant
                   \raw_encoding\font_width,
                 \raw_encoding>
                <\raw_encoding>
                <\raw_encoding>
                <\latex_family>
                <\latex_weight\latex_width>
                <\latex_shape>}
⟨/debug⟩
             \installrawfont{
                \font_family\font_weight\font_shape\raw_variant
                   \raw_encoding\font_width
             }{ \font_family\font_weight\font_shape\raw_variant
                   \raw_encoding\font_width,
                \raw_encoding
             }{ \raw_encoding
             }{ \raw_encoding
             }{ \latex_family
             }{ \latex_weight\latex_width
```

```
            }{ \latex_shape
            }{}
         \fi
      \fi
   \fi
}
```

`\fake_shape_o_expert_encoding` Called from `\fake_shape_o` if the oblique shape needs to be faked. Invokes

```
\transformfont{<FONT>o8x}
  {\slantfont{SLANT}{\frommtx{<FONT>8x}}}
```

to generate an 8x-reencoded `.mtx` file (and a raw `.pl` file).

```
\def\fake_shape_o_expert_encoding{
   \if_file_exists{
      \font_family\font_weight\raw_variant
         \expert_encoding\font_width.mtx
   }\then
      \if_file_exists{
         \font_family\font_weight\font_shape\raw_variant
            \expert_encoding\font_width.mtx
      }\then % no action required
      \else
⟨*debug⟩
         \immediate\write16{
            INFO>~run~\string\transformfont\space\space
            <\font_family\font_weight\font_shape\raw_variant
               \expert_encoding\font_width>~from~
            <\font_family\font_weight\raw_variant
               \expert_encoding\font_width>~(faking~oblique)
         }
⟨/debug⟩
         %%% WARNING: famtool.pl relies on this message format!!!
         \immediate\write16{
            Faking~oblique~font~
            \font_family\font_weight\font_shape\raw_variant
               \expert_encoding\font_width
            \space from~
            \font_family\font_weight\raw_variant
               \expert_encoding\font_width
         }
         \transformfont{
            \font_family\font_weight\font_shape\raw_variant
               \expert_encoding\font_width
         }{
            \slantfont{\SlantAmount}{
               \frommtx{
                  \font_family\font_weight\raw_variant
                     \expert_encoding\font_width
               }
            }
         }
      \fi
   \fi
}
```

`ake_shape_o_oldstyle_encoding` Called from `\fake_shape_o` if the oblique shape needs to be faked. Invokes

```
\transformfont{<FONT>oj8r}
  {\slantfont{SLANT}{\frommtx{<FONT>j8r}}}
```

to generate an `8r`-reencoded `.mtx` file (and a raw `.pl` file).

```
⟨*oldstyle⟩
\def\fake_shape_o_oldstyle_encoding{
   \if_file_exists{
      \font_family\font_weight\oldstyle_variant
         \raw_encoding\font_width.mtx
   }\then
      \if_file_exists{
         \font_family\font_weight\font_shape\oldstyle_variant
            \raw_encoding\font_width.mtx
      }\then % no action required
      \else
⟨*debug⟩
         \immediate\write16{
            INFO>~run~\string\transformfont\space\space
            <\font_family\font_weight\font_shape\oldstyle_variant
               \raw_encoding\font_width>~from~
            <\font_family\font_weight\oldstyle_variant
               \raw_encoding\font_width>~(faking~oblique)
         }
⟨/debug⟩
         %%% WARNING: famtool.pl relies on this message format!!!
         \immediate\write16{
            Faking~oblique~font~
            \font_family\font_weight\font_shape\oldstyle_variant
               \raw_encoding\font_width
            \space from~
            \font_family\font_weight\oldstyle_variant
               \raw_encoding\font_width
         }
         \transformfont{
            \font_family\font_weight\font_shape\oldstyle_variant
               \raw_encoding\font_width
         }{
            \slantfont{\SlantAmount}{
               \frommtx{
                  \font_family\font_weight\oldstyle_variant
                     \raw_encoding\font_width
               }
            }
         }
      \fi
   \fi
}
⟨/oldstyle⟩
```

`\fake_shape_i` Call `\fake_shape_` to generate an `8r`-reencoded `.mtx` file and a ligfull `.pl` file, if a corresponding `8a`-encoded `.afm` file exists. Otherwise, do nothing, since an italic shape can't be faked.

```
\let\fake_shape_i\fake_shape_  % We must do this again!
```

17.3 Faking font widths

\if_fake_narrow_

```
\newif\if_fake_narrow_
\_fake_narrow_false
```

\fakenarrow `\fakenarrow{WIDTH}`

Sets the expansion factor used to generate faked narrow fonts. If it isn't set, do not attempt to install faked narrow fonts.

```
\def\fakenarrow#1{
   \_fake_narrow_true
   \gdef\fake_narrow_width{#1}
}
```

\fake_width_ Do nothing for the default width or the condensed width.

```
\def\fake_width_{}
\def\fake_width_c{}
```

\fake_width_n If we are faking narrow fonts, check if an `8a`-encoded `.afm` file exists for the current shape in narrow width, and if not call `\fake_shape_n_raw_encoding` and `\fake_shape_n_expert_encoding`.

```
\def\fake_width_n{
   \if_fake_narrow_
      \if_file_exists{
         \font_family\font_weight\raw_shape\raw_variant
            \adobe_encoding\font_width.afm
      }\then % no action required
      \else
         \fake_width_n_raw_encoding
        \if_expert_
         \fake_width_n_expert_encoding
        \fi
      \fi
   \fi
}
```

\fake_width_n_raw_encoding Called from `\fake_width_n` if the narrow width needs to be faked. Invokes

```
\transformfont{<FONT>8rn}
  {\xscalefont{WIDTH}{\frommtx{<FONT>8r}}}
```

to generate an `8r`-encoded `.mtx` file for a faked narrow font, followed by

```
\installrawfont{<FONT>8rn}{<FONT>8rn,8r}{8r}...
```

to install a ligfull `.pl` file.

```
\def\fake_width_n_raw_encoding{
   \if_file_exists{
      \font_family\font_weight\font_shape\raw_variant
         \raw_encoding.mtx
   }\then
```

```
      \if_file_exists{
         \font_family\font_weight\font_shape\raw_variant
            \raw_encoding\font_width.mtx
      }\then % no action required
      \else
⟨*debug⟩
         \immediate\write16{
            INFO>~run~\string\transformfont\space\space
            <\font_family\font_weight\font_shape\raw_variant
               \raw_encoding\font_width>~from~
            <\font_family\font_weight\font_shape\raw_variant
               \raw_encoding>~(faking~narrow)
         }
⟨/debug⟩
         %%% WARNING: famtool.pl relies on this message format!!!
         \immediate\write16{
            Faking~narrow~font~
            \font_family\font_weight\font_shape\raw_variant
               \raw_encoding\font_width
            \space from~
            \font_family\font_weight\font_shape\raw_variant
               \raw_encoding
         }
         \transformfont{
            \font_family\font_weight\font_shape\raw_variant
               \raw_encoding\font_width
         }{
            \xscalefont{\fake_narrow_width}{
               \frommtx{
                  \font_family\font_weight\font_shape\raw_variant
                     \raw_encoding
               }
            }
         }
         \if_file_exists{ \raw_encoding.mtx }\then
⟨*debug⟩
            \immediate\write16{
               INFO>~run~\string\installrawfont\space
               <\font_family\font_weight\font_shape\raw_variant
                  \raw_encoding\font_width>
               <\font_family\font_weight\font_shape\raw_variant
                  \raw_encoding\font_width,
                \raw_encoding>
               <\raw_encoding>
               <\raw_encoding>
               <\latex_family>
               <\latex_weight\latex_width>
               <\latex_shape>}
⟨/debug⟩
            \installrawfont{
               \font_family\font_weight\font_shape\raw_variant
                  \raw_encoding\font_width
            }{ \font_family\font_weight\font_shape\raw_variant
                  \raw_encoding\font_width,
```

```
               \raw_encoding
            }{ \raw_encoding
            }{ \raw_encoding
            }{ \latex_family
            }{ \latex_weight\latex_width
            }{ \latex_shape
            }{}
         \fi
      \fi
   \fi
}
```

`\fake_width_n_expert_encoding` Called from `\fake_width_n` if the narrow width needs to be faked. Invokes

```
\transformfont{<FONT>8xn}
  {\xscalefont{WIDTH}{\frommtx{<FONT>8x}}}
```

to generate an 8x-encoded `.mtx` file for a faked narrow font.

```
\def\fake_width_n_expert_encoding{
   \if_file_exists{
      \font_family\font_weight\font_shape\raw_variant
         \expert_encoding.mtx
   }\then
      \if_file_exists{
         \font_family\font_weight\font_shape\raw_variant
            \expert_encoding\font_width.mtx
      }\then % no action required
      \else
⟨*debug⟩
         \immediate\write16{
            INFO>~run~\string\transformfont\space\space
            <\font_family\font_weight\font_shape\raw_variant
               \expert_encoding\font_width>~from~
            <\font_family\font_weight\font_shape\raw_variant
               \expert_encoding>~(faking~narrow)
         }
⟨/debug⟩
         %%% WARNING: famtool.pl relies on this message format!!!
         \immediate\write16{
            Faking~narrow~font~
            \font_family\font_weight\font_shape\raw_variant
               \expert_encoding\font_width
            \space from~
            \font_family\font_weight\font_shape\raw_variant
               \expert_encoding
         }
         \transformfont{
            \font_family\font_weight\font_shape\raw_variant
               \expert_encoding\font_width
         }{
            \xscalefont{\fake_narrow_width}{
               \frommtx{
                  \font_family\font_weight\font_shape\raw_variant
                     \expert_encoding
```

```
            }
          }
        }
      \fi
    \fi
}
```

17.4 Installing reencoded fonts

\latin_encoding
\font_encoding
\latex_encoding
\latex_mtx

`\latin_encoding{FONTENC}{EXPERTISED-ENC}{OLDSTYLE-ENC}{LATEXENC}{LATEXMTX}`

If this is an expertised family `EXPERTISED-ENC` is used instead of `FONTENC`. If this is an expertised family with oldstyle digits `OLDSTYLE-ENC` is used instead of `EXPERTISED-ENC`.

UV 06/98: Added code for oldstyle encodings from Alan's v 1.511. The `\font_encoding` is changed when `\if_oldstyle_` is true.

UV 06/98: Added anonther parameter for the default metrics file (which is usually either `latin.mtx` or `textcomp.mtx`).

UV 06/98: Added experimental code for OsF or SC&OsF fonts.

```
\def\latin_encoding#1#2#3#4#5{
  \gdef\latex_encoding{#4}
  \gdef\latex_mtx{#5}
  \if_oldstyle_
    \gdef\font_encoding{#3}
  \else\if_expert_
    \gdef\font_encoding{#2}
  \else
    \gdef\font_encoding{#1}
  \fi\fi
  \gdef\expert_font{}
  \gdef\oldstyle_font{}
  \if_expert_
    \if_file_exists{
      \font_family\font_weight\raw_shape\raw_variant
        \expert_encoding\font_width.mtx
    }\then
      \gdef\expert_font{
        \font_family\font_weight\raw_shape\raw_variant
          \expert_encoding\font_width,
      }
    \fi
  \fi
⟨*oldstyle⟩
  \if_oldstyle_
    \if_file_exists{
      \font_family\font_weight\raw_shape\oldstyle_variant
        \raw_variant\raw_encoding\font_width.mtx
    }\then
      \gdef\oldstyle_font{
        unsetnum,
        \font_family\font_weight\raw_shape\oldstyle_variant
          \raw_variant\raw_encoding\font_width,
        resetosf,
      }
```

```
      \fi
   \fi
⟨/oldstyle⟩
⟨*debug⟩
   \immediate\write16{
      INFO>~run~\string\installfont\space
      <\font_family\font_weight\font_shape\font_variant
         \font_encoding\font_width>
      <\font_family\font_weight\raw_shape\raw_variant
         \raw_encoding\font_width,
      \if_oldstyle_
       \ifx\raw_shape\caps_shape
         resetosf,
       \fi
      \fi
      \expert_font
      \oldstyle_font
      \latex_mtx>
      <\latex_encoding\encoding_shape>
      <\latex_encoding>
      <\latex_family>
      <\latex_weight\latex_width>
      <\latex_shape>}
⟨/debug⟩
   \installfont{
      \font_family\font_weight\font_shape\font_variant
         \font_encoding\font_width
   }{
      \font_family\font_weight\raw_shape\raw_variant
         \raw_encoding\font_width,
      \if_oldstyle_
       \ifx\raw_shape\caps_shape
         resetosf,
       \fi
      \fi
      \expert_font
      \oldstyle_font
      \latex_mtx
   }{
      \latex_encoding\encoding_shape
   }{
      \latex_encoding
   }{
      \latex_family
   }{
      \latex_weight\latex_width
   }{
      \latex_shape
   }{}
}
```

17.5 Default weights, widths and shapes

Fontname: `weight.map`		NFSS: *LaTeX Companion*, p. 190	
`a`	Thin Hairline	`ul`	Ultra Light
`j`	ExtraLight	`el`	Extra Light
`l`	Light	`l`	Light
`r`	Regular Roman	`m`	Medium
`k`	Book	`m`	Medium
`m`	Medium	`mb`	(was: `m`)
`d`	Demi	`db`	(was: `sb`)
`s`	Semibold	`sb`	Semibold
`b`	Bold	`b`	Bold
`h`	Heavy Heavyface	`eb`	(was missing)
`c`	Black	`eb`	(was missing)
`x`	ExtraBold ExtraBlack	`eb`	Extra Bold
`u`	Ultra UltraBlack	`ub`	Ultra Bold
`p`	Poster		(still missing)

`\latin_weights` Each call to `\latin_weight` maps a Fontname weight code (`\font_weight`) to a LaTeX weight code (`\latex_weight`). Non-existing weights are ignored (or substituted when the `.fd` files are written out by `\endinstallfont`).

The standard values are given in the table. They may be changed, but you'd better know what you're doing.

UV, 04/98: Changed the processing order: Do the most common shapes first. Added new mappings for `c` and `h`, changed mapping for `m` to newly invented LaTeX weight `mb`.

```
\def\latin_weights{
   \latin_weight{r}{m}
   \latin_weight{k}{m}
   \latin_weight{b}{b}
   \latin_weight{s}{sb}
   \latin_weight{d}{db} % was {d}{sb}, SPQR changed
   \latin_weight{m}{mb} % was {m}{m},  UV changed
   \latin_weight{c}{eb} % UV added
   \latin_weight{h}{eb} % UV added
   \latin_weight{x}{eb}
   \latin_weight{u}{ub}
   \latin_weight{l}{l}
   %\latin_weight{j}{el}
   %\latin_weight{a}{ul} % UV added
}
```

Fontname: `width.map`		NFSS: *LaTeX Companion*, p. 190	
t	Thin	–	–
o	Ultra Condensed	uc	Ultra Condensed
u	Ultra Compressed	uc	. .
q	Extra Compressed	ec	Extra Condensed
c	Condensed	c	Condensed
p	Compressed	c	. .
n	Narrow	c	. .
–	–	sc	Semi Condensed
r	Normal, Medium, Regular	m	Medium
–	–	sx	Semi Expanded
e	Expanded	x	Expanded
x	Extended	x	. .
v	Extra Expanded	ex	Extra Expanded
–	–	ux	Ultra Expanded
w	Wide	–	–

`\latin_widths` Each call to `\latin_width` maps a Fontname width code (`\font_width`) to a LaTeX width code (`\latex_width`). Non-existing narrow fonts are faked only if `\fakenarrow` is specifically called for.

```
\def\latin_widths{
   \latin_width{}{}
   \latin_width{n}{c}
   %latin_width{c}{c}
   %latin_width{x}{x}
}
```

`\latin_shapes` The fifth argument of `\latin_shape` is stored in the variable `\encoding_switches`.

```
\def\latin_shapes{
   \latin_shape{} {} {} {n} {\_textcomp_true }
   \latin_shape{c}{c}{} {sc}{\_textcomp_false}
   \latin_shape{o}{o}{} {sl}{\_textcomp_true }
   \latin_shape{i}{i}{i}{it}{\_textcomp_true }
}
```

`\latin_encodings` `\encoding_switches` is evaluated immediatley before the `\if_textcomp_` test to set it as appropriate for the curent shape.

```
\def\latin_encodings{
   \latin_encoding{7t}{9t}{9o}{OT1}{latin}
   \latin_encoding{8t}{9e}{9d}{T1} {latin}
⟨*textcomp⟩
   \encoding_switches
   \if_textcomp_
      \latin_encoding{8c}{9c}{9c}{TS1}{textcomp}
   \fi
⟨/textcomp⟩
}
```

`\if_textcomp_` Switch to control whether or not to install a text companion encoding. It is set or unset for each shape by `\latin_shapes` and evaluated in `\latin_encodings`.

```
\newif\if_textcomp_
```

```
  \raw_encoding
 \adobe_encoding
\expert_encoding
\def\raw_encoding{8r}
\def\adobe_encoding{8a}
\def\expert_encoding{8x}

    \caps_shape
 \oldstyle_shape
\def\caps_shape{c}
\def\oldstyle_shape{j}

   \SlantAmount
\def\SlantAmount{167}
⟨/pkg⟩
```

File e
fimapgen.dtx

18 Generating map file fragments

A *map file* is a file which is used by some DVI driver to link various kinds of font information to each other, as is needed for the driver's operation. Examples of such files are the file `psfonts.map` used by `dvips` and the config files of OzTEX. There are however many other such files around in the TEX world.

Not all of a map file need to be related to linking specific kinds of font information to each other. OzTEX config files can for example be used to set the values of TEX implementation parameters[6] and there need not be any font-related information at all in them. The important thing is however that there can be map information in them—that information could be automatically written by the routines described here.

Since there may be other kinds of information in these files (and often enough need to be for everthing to work right), the routines described here will only generate map file *fragments*. These fragments will then have to be inserted in the final map files somehow, usually through manual editing. The purpose of the routines described here is only to save the user a good deal of (often quite boring) typing, not to actually install everything where it needs to be.

`\mapfrags_subsystem` This macro holds the name of the automatic map file fragments writer subsystem of fontinst, as it appears in error messages.

```
1 \def\mapfrags_subsystem{Map~fragments~writer}
```

18.1 Interface to main fontinst

The automatic generation of map file fragments is based on the two commands `\storemapdata` and `\makemapentry`. Their respective syntaxes are

`\storemapdata{`⟨*TEX font name*⟩`}{`⟨*source*⟩`}{`⟨*transforms*⟩`}`
`\makemapentry{`⟨*TEX font name*⟩`}`

`\storemapdata` `\storemapdata` commands are written when a metric file ⟨*TEX font name*⟩`.mtx` is generated. The purpose of these commands is to record where the data for that file came from, and what was done to these data. The source of the data is specified in the ⟨*source*⟩ argument, which can contain one of

`\fromafm{`⟨*AFM name*⟩`}{`⟨*PS name*⟩`}`
`\frompl{`⟨*PL name*⟩`}`
`\frommtx{`⟨*MTX name*⟩`}`
`\fromvpl`

There is a difference between `\fromvpl` and the first three in that virtual fonts need no map file entry (and hence there should not be any either). Besides some error checking, `\fromvpl` causes a `\makemapentry` to do essentially nothing.

What was done to the data is specified by the ⟨*transforms*⟩ argument, which is a possibly empty sequence of commands, in which each item is one of

[6] The kind of things that were Pascal constants in the original WEB sources for TEX.

```
\transformfont{⟨x-scale⟩}{⟨slant-scale⟩}{⟨y-scale⟩}
\reencodefont{⟨etx⟩}
\reglyphfont
```

The order is interpreted so that the thing done first appear first in the sequence. In normal cases there is at most one of each, and then the order is not important. ⟨*x-scale*⟩, ⟨*slant-scale*⟩, and ⟨*y-scale*⟩ are the values of these variables used for the transformation, as TEX numbers. ⟨*etx*⟩ is the name of the ETX file used for the font reencoding. `\reglyphfont` is an error marker. It will not be encountered (but it may well get stored in some table) unless a metric font transform is applied to a font that has been reglyphed, which is illegal.

`\makemapentry` `\makemapentry` commands are written when a VPL that uses or a ligfull PL for a font is generated. These commands are the ones which actually cause the map file fragment generator to write an entry, but entries are only written if they haven't been written before. `\makemapentry` uses information that was stored by a previous call to `\storemapdata`.

`\storemapdata` stores its arguments in one of the control sequences `\Tf-`⟨*font*⟩, where ⟨*font*⟩ is the same thing as ⟨*TEX font name*⟩ above. `Tf` stands for "TEX font". The control sequences in this family are paramterless macros whose replacement texts have the following form:

```
{⟨source⟩}{⟨transforms⟩}{⟨made⟩}
```

Here ⟨*source*⟩ and ⟨*transforms*⟩ are exactly as above. ⟨*made*⟩ is either `\if_false` or `\if_true` in case a map file entry for the font in question has been written or not, respectively. These control sequences should always be set globally.

`\recordtransforms` In main fontinst, the command

`\endrecordtransforms` `\transform_record_file`

```
\recordtransforms{⟨filename⟩}
```

starts a block of code within which font transforms will be recorded in the file ⟨*filename*⟩. `\endrecordtransforms` ends such a block.

```
⟨*pkg⟩
\def\recordtransforms#1{\open_pout\transform_record_file{#1}}
\def\endrecordtransforms{
   \close_pout\transform_record_file{Font~transformation~records}
}
\chardef\transform_record_file=\closed_stream
⟨/pkg⟩
```

`\storemapdata` There should not be two `\storemapdata` for the same font, hence the check below.

```
⟨*map⟩
\def\storemapdata#1#2#3{
   \x_cs\ifx{Tf-#1}\x_relax \else
      \fontinstwarning\mapfrags_subsystem{Font~#1~generated~again}
   \fi
   \x_cs\gdef{Tf-#1}{{#2}{#3}\if_true}
}
⟨/map⟩
```

`\record_transform` In main fontinst, the call

```
\record_transform{⟨TFM name⟩}{⟨source⟩}{⟨transforms⟩}
```

writes

\storemapdata{⟨*TFM name*⟩}{⟨*source*⟩}{⟨*transforms*⟩}

to the transformation recordings file, if font transforms are being recorded. Note that the caller of \record_transform must see to that macros in ⟨*source*⟩ and ⟨*transforms*⟩ are written correctly to the file (this is usually accomplished by adding \string in suitable places of these arguments during their construction).

```
⟨*pkg⟩
\def\record_transform#1#2#3{
   \ifnum \transform_record_file=\closed_stream \else
      \pout_line\transform_record_file{\string\storemapdata{#1}{#2}{#3}}
   \fi
}
⟨/pkg⟩
```

\transformfont
\reencodefont
\reglyphfont

These are initially \relax, so they can be \edefed with safely.

```
⟨*map⟩
\let\transformfont\x_relax
\let\reencodefont\x_relax
\let\reglyphfont\x_relax
```

\makemapentry

\makemapentry sets \TeX_font_name, checks if any map data have been stored for the font in question, and passes them on to \make_map_entry if they have; otherwise it raises an error. There is also a group which is begun and ended by \makemapentry, to stop values deduced for one entry to interfere with the deduction of values in the following.

```
\def\makemapentry#1{\begingroup
   \def\TeX_font_name{#1}
   \expandafter\let \expandafter\a_macro \csname Tf-#1\endcsname
   \ifx\a_macro\x_relax
      \from_unknown{#1}
      \x_cs\gdef{Tf-#1}{{\from_unknown{#1}}{}\if_false}
   \else
      \expandafter\make_map_entry\a_macro
   \fi
   \endgroup
}
```

\make_map_entry

\make_map_entry oversees the interpretation of the ⟨*source*⟩ part of the map data, something which sets \source_font_TeX_name and possibly \PS_font_name. It also stores the transforms in \font_transforms, calls a list macro \entry_makers_list which causes the actual entries to be written, and records in \Tf-⟨*TEX font name*⟩ that the entries have been written.

```
\def\make_map_entry#1#2#3{
   #3
      \def\font_transforms{#2}
      #1
      \begingroup
         \_a_false
         \let\transformfont\gobble_three
         \let\reencodefont\gobble_one
         \let\reglyphfont\_a_true
```

```
48          \font_transforms
49       \expandafter\endgroup \if_a_
50          \fontinsterror\mapfrags_subsystem{
51             \string\makemapentry\space for~reglyphed~font
52          }{ No~entry~for~font~\TeX_font_name\space
53             can~be~written,\messagebreak
54             since~it~had~been~reglyphed!~\error_help_a
55          }
56       \else
57          \entry_makers_list
58       \fi
59       \x_cs\gdef{Tf-\TeX_font_name}{{#1}{#2}\if_false}
60    \fi
61 }
62 ⟨/map⟩
```

`\record_usage` In main fontinst, the call

`\record_usage{`⟨*TFM name*⟩`}`

writes

`\makemapentry{`⟨*TFM name*⟩`}`

to the transformation recordings file, if font transforms are being recorded. The `\record_usage` command is, roughly speaking, only used by `\installfont` and `\installrawfont`.

```
63 ⟨*pkg⟩
64 \def\record_usage#1{
65    \ifnum \transform_record_file=\closed_stream \else
66       \pout_line\transform_record_file{\string\makemapentry{#1}}
67    \fi
68 }
69 ⟨/pkg⟩
```

`\fromafm` `\fromafm` saves its two arguments in the suitable string macros.

```
70 ⟨*map⟩
71 \def\fromafm#1#2{
72    \def\source_font_TeX_name{#1}
73    \def\PS_font_name{#2}
74 }
```

`\frompl` `\frompl` saves its argument in the suitable string macro.

```
75 \def\frompl#1{
76    \def\source_font_TeX_name{#1}
77 }
```

`\frommtx` This one is tricky, since it will have to work recursively, fetching data that were stored for some other font.

```
78 \def\frommtx#1{%
79    \expandafter\let\expandafter\a_macro \csname Tf-#1\endcsname
80    \ifx \a_macro\x_relax
81       \from_unknown{#1}
82       \x_cs\gdef{Tf-#1}{{\from_unknown{#1}}{}\if_true}
```

```
   \else
      \expandafter\from_mtx \a_macro
   \fi
}
```

`\from_mtx`

```
\def\from_mtx#1#2#3{
   \edef\font_transforms{#2\font_transforms}
   #1
}
```

`\fromvpl` The `\fromvpl` command checks if `\font_transforms` is empty. If it is then everything is fine, but no entry should be written for this font. If it isn't then someone has tried to make a transformation of a virtual font, which doesn't work.

```
\def\fromvpl{
   \ifx \empty_command\font_transforms \else
      \fontinsterror\mapfrags_subsystem{
         Font~is~virtual,~it~cannot~be~transformed}{
         No~entry~for~font~\TeX_font_name\space can~be~written.
         \messagebreak\error_help_a
      }
   \fi
   \let\maker_do\gobble_one
}
```

`\AssumeMetafont` The `\AssumeMetafont` general settings command makes `\frompl` behave like `\fromvpl`.

```
\def\AssumeMetafont{
   \def\frompl{
      \ifx \empty_command\font_transforms \else
         \fontinsterror\mapfrags_subsystem{
            A~Metafont~cannot~be~transformed}{
            No~entry~for~font~\TeX_font_name\space can~be~written.
            \messagebreak\error_help_a
         }
      \fi
      \let\maker_do\gobble_one
   }
}
```

`\from_unknown` This special value for ⟨*source*⟩ is used for fonts that are referenced by some `\frommtx` but which have no `\storemapdata` themselves.

```
\def\from_unknown#1{
   \fontinstwarningnoline\mapfrags_subsystem{
      No~map~data~stored~for~font~#1.\messagebreak
      No~entry~for~\TeX_font_name\space will~be~written,\messagebreak
      due~to~insufficient~information}
   \global\G_unknown_source_true
   \let\maker_do\gobble_one
}
```

18.2 User interface

The basic usage of the automatic map file fragment generator consists of specifying for which driver(s) entries should be generated, `\input`ing a file of recorded transformations generated by main fontinst, and signaling that there isn't anything more to write. This can be done with the commands

> `\adddriver{`⟨*driver name*⟩`}{`⟨*fragment file name*⟩`}`
> ⟨*possible additional* `\adddriver` *commands*⟩
> `\input` ⟨*recorded transforms file*⟩
> `\donedrivers`
> `\bye`

It should however be noted that this is the *basic* usage. I imagine that there should be lots of ways in which the texts written can be configured—the various settings that might be needed on different systems are simply to diverse for one default setting to work everywhere! Some questions for starters are (and these are for dvips alone; that's the driver I know best):

- Is the font resident on the printer or must it be downloaded?
- If the font must be downloaded, what is the name of the file it is in? Here one must take into account not only the possibility that the font file might have to be shared with some other software (such as a GUI) and thus cannot be named according to the Berry scheme—even if it is named according to the Berry scheme one still has the question of its extension: `.pfa`, `.pfb`, or something else? For instances of Multiple Master fonts, you (at least sometimes) have to download more than one file.
- Again if the font must be downloaded, can/should it be partially downloaded?
- In case the font has been reencoded, what is the name of the file which defines this encoding, and by what name does that file make the encoding known to the postscript interpreter? This question is rather easily solved by defining a table which lists a postscript encoding file name and object name for each ETX file, but there must be a user level interface for entering additional data in this table, in case the user defines an encoding of his or hers own.

I suspect map files for DVI viewers are on average more complicated than map files for DVI printer drivers, since the latter are more likely to be restricted to platform-specific font names.

An *entry maker* is a group of macros which write the entry for the font currently under consideration to a map fragments file each time they are called. The macro to call must have a name of the form `\make_`⟨*driver*⟩ for `\adddriver` to recognise it. Each entry maker has a separate output file (opened using `\open_pout`) and the identifier connected to that file is called `\output_`⟨*driver*⟩.

A list of all entry makers currently active is maintined in the macro `\entry_makers_list`. Each item in this list has the form

> `\maker_do \make_`⟨*driver*⟩

\maker_do is usually \relax, but it does occationally get set to other values.

\adddriver The \adddriver command adds the named driver to the list of drivers to write entries for and opens an output file where the entries for that particular driver will go. Repeated calls to \adddriver for the same driver have no effect.

```
\def\adddriver#1#2{
   \x_cs\ifx{make_#1}\x_relax
      \fontinsterror\mapfrags_subsystem
         {There~is~no~entry~maker~for~#1}\error_help_a
   \else
      \x_cs\ifx{output_#1}\x_relax \_a_true \else
         \x_cs\ifx{output_#1}\closed_stream \_a_true \else
            \_a_false
      \fi\fi
      \if_a_
         \x_cs\open_pout{output_#1}{#2}
         \expandafter\add_to \expandafter\entry_makers_list
            \expandafter{ \expandafter\maker_do
            \csname make_#1\endcsname}
      \fi
   \fi
}
```

\entry_makers_list
\maker_do

```
\let\entry_makers_list\empty_command
\let\maker_do\x_relax
```

\donedrivers The main function performed by the \donedrivers command is to close all the open output files. It might also prints some warning messages.

```
\def\donedrivers{
   \def\maker_do##1{
      \x_cs\close_pout{output\expandafter\gobble_five\string##1}
         {Map~file~fragments}
   }
   \entry_makers_list
   \let\maker_do\x_relax
   \let\entry_makers_list\empty_command
   \errorstopmode
   \ifG_unknown_source_
      \fontinstwarningnoline\mapfrags_subsystem{
         Some~font~is~missing~from~the~output~file(s),\messagebreak
         since~its~source~is~unknown}
   \fi
   \global\G_unknown_source_false
   \if_undecided_
      \fontinstwarningnoline\mapfrags_subsystem{
         Check~the~output~file(s)--some~data~could~not~be~determined}
   \fi
   \_undecided_false
   \if_uncertain_
      \fontinstwarningnoline\mapfrags_subsystem{
         Check~the~output~file(s)--some~data~was~considered~uncertain}
```

```
    \fi
    \_uncertain_false
}
```

\gobble_five

```
\def\gobble_five#1#2#3#4#5{}
```

\if_undecided_ _undecided_true _undecided_false

This switch should be made true if a \clueless_str is written to some of the output files.

```
\def\_undecided_true{\global\let\if_undecided_\iftrue}
\def\_undecided_false{\global\let\if_undecided_\iffalse}
\_undecided_false
```

\if_uncertain_ _uncertain_true _uncertain_false

A value-determining macro can set this switch to true if it thinks that the value it just set its string to has a fair chance of being wrong. Hopefully, it shouldn't have to be used too often.

```
\def\_uncertain_true{\global\let\if_uncertain_\iftrue}
\def\_uncertain_false{\global\let\if_uncertain_\iffalse}
\_uncertain_false
```

\ifG_unknown_source_ \G_unknown_source_true \G_unknown_source_false

If at some point the named source for a font is unknown, this switch should be set to true. The G is because it should be set globally.

```
\newif\ifG_unknown_source_
```

18.3 Deducing values for the map file entries

18.3.1 Basic principles

The automatic generation of map file fragments mainly consists of determining various text strings and writing these to a file as the format of the map file being written requires. The complicated part is usually to determine what these text strings should be, since the only restriction imposed by the output format often is that the strings should be written in the right order and with separating whitespace.

It is furthermore the case that some of these text strings seem to be hard to deduce from the information primarily given in the command to write the entry, but somewhat easier to deduce from some other deduced text string.

Therefore the following model seems appropriate. For each text string that may need to be deduced, there is one macro which is used to store the string (if it has been deduced) and one macro which can be called to set the former macro to an appropriate value. Thus there is one macro \PS_font_name which stores the postscript name for the font currently in question and one macro \get_PS_font_name which determines the proper value for \PS_font_name.

\unknown_str

There is also a macro \unknown_str to which all the string storing macros should initially be \let, so that it can be easily tested whether a string storing macro has been given its proper value or not.

Typically, if \PS_font_name is used somewhere, one would first have a piece of code that says

```
\ifx\PS_font_name\unknown_str
   \get_PS_font_name
\fi
```

to ensure that it is known when it is used.

`\unknown_str` A suitable value for this macro seems to be `?????`.

```
\def\unknown_str{?????}
```

`\clueless_str` To further support this model, there is also a macro `\clueless_str` which a string storing macro should be `\let` equal to if its proper value could not be determined. This distinction is useful for the `\get_`... macros, since if `\get_`A can determine the correct value of A from B, but also from C, then it helps if `\get_`A can easily check whether it has previously turned out to be impossible to determine the value of B, because then it can simply deduce A from C instead.

```
\edef\clueless_str{\unknown_str ?}
```

An example of how `\clueless_str` should be used is the following skeleton of a `\get_`A macro:

```
\def\get_A{
   \ifx\unknown_str\B \get_B \fi
   \ifx\clueless_str\B
      \ifx\unknown_str\C \get_C \fi
      \ifx\clueless_str\C
         \let\A\clueless_str
      \else
         ⟨Deduce A from C⟩
      \fi
   \else
      ⟨Deduce A from B⟩
   \fi
}
```

18.3.2 Implementation

`\TeX_font_name` This is the name (as appearing in a DVI or VF file) of the font for which an entry should be made. It is set by `\makemapentry`, so no `\get_`... macro is needed.

```
\let\TeX_font_name\unknown_str
```

`\source_font_TeX_name` This is the name (as AFM or PL) of the font whose metrics were used by fontinst. It is set by `\fromafm` or `\frompl`, so no `\get_`... macro is needed.

```
\let\source_font_TeX_name\unknown_str
```

`\PS_font_name` This is the name (as FontName entry in an AFM or correspondingly) of the font in question.

```
\let\PS_font_name\unknown_str
```

`\get_PS_font_name` I expect this to become one of the central macros in the map file generation (at least when entries are generated for postscript fonts), but for the moment it does the closest thing to nothing a `\get_`... macro is allowed to do; it admits it hasn't got a clue about what the proper value is.

```
\def\get_PS_font_name{\let\PS_font_name\clueless_str}
```

\AssumeAMSBSYY
\AssumeBaKoMa

These commands redefine \get_PS_font_name so that it returns the PS name this font would have if it was part of the AMS/Blue Sky/Y&Y and BaKoMa respectively distribution of the Computer Modern fonts—the TEX font name in all upper case and all lower case respectively.

```
\def\AssumeAMSBSYY{\def\get_PS_font_name{
   \expandafter\uppercase \expandafter{ \expandafter\def
      \expandafter\PS_font_name \expandafter{ \source_font_TeX_name}
   }
}}
\def\AssumeBaKoMa{\def\get_PS_font_name{
   \expandafter\lowercase \expandafter{ \expandafter\def
      \expandafter\PS_font_name \expandafter{ \source_font_TeX_name}
   }
}}
```

\font_transforms

This is argument #3 of \makemapentry. It is not a string, so its default value is not \unknown_str.

```
\let\font_transforms\empty_command
```

18.4 Driver makers

This subsection contains the definitions of macros that makes up the driver makers.

18.4.1 The debug driver

The debug driver maker doesn't really generate entries for any DVI driver map file, it just prints the data that was available. It is mainly intended for debugging purposes, but it might also serve as a basic model for other driver makers.

\make_debug

```
\def\make_debug{
   \pout_line\output_debug{Driver~data~for~font~\TeX_font_name:}
   \pout_lline\output_debug{Source~font~TeX~name:~\source_font_TeX_name}
   \ifx \font_transforms\empty_command
      \pout_lline\output_debug{No~font~transforms~applied.}
   \else
      \pout_lline\output_debug{Font~transforms~applied:}
      \bgroup
         \let\reencodefont\debug_reencodefont
         \let\transformfont\debug_transformfont
         \let\reglyphfont\debug_reglyphfont
         \font_transforms
      \egroup
   \fi
   \debug_values_hook
}
```

\debug_reencodefont

```
\def\debug_reencodefont#1{
   \pout_llline\output_debug{Font~reencoded~using~#1.etx.}
}
```

\debug_transformfont

```
\def\debug_transformfont#1#2#3{
   \pout_llline\output_debug{Font~transformed~by~(
      (\make_factor{#1}~\make_factor{#2})~(0.000~\make_factor{#3})).}
}
```

\debug_reglyphfont Remember that executing \reglyphfont is a sign that something is wrong.

```
\def\debug_reglyphfont{
   \pout_llline\output_debug{ERROR!~Reglyphed~fonts~shouldn't~
      appear~here!}
   \fontinsterror\mapfrags_subsystem{
      \string\reglyphfont\space encountered~among~font~transforms}
      \error_help_d
}
```

\debug_value The \debug_value macro can be used to print the name and contents of an arbitrary value macro in the debug driver. It is called with the value macro as argument, e.g.

```
\debug_value\PS_font_name
```

```
\def\debug_value#1{
   \ifx #1\unknown_str
      \csname get_ \expandafter\gobble_one \string#1 \endcsname
   \fi
   \pout_lline\output_debug{\string#1:~#1}
   \ifx #1\clueless_str \_undecided_true \fi
}
```

\debugvalue \debug_values_hook The \debugvalue command is a user level interface for adding value macros to the list of those that are displayed by the debug driver. The argument is the name of the value in question, e.g.

```
\debugvalue{PS_font_name}
```

Note that the name given to \debugvalue is not the control sequence in which the name is stored.

```
\def\debugvalue#1{
   \expandafter\add_to \expandafter\debug_values_hook
   \expandafter{ \expandafter\debug_value \csname#1\endcsname}
}
```

```
\let\debug_values_hook\empty_command
\debugvalue{PS_font_name}
```

⟨/map⟩

Index

Numbers written in italic refer to the page where the corresponding entry is described, the ones underlined to the code line of the definition, the rest to the code lines where the entry is used.

File Key: a=fibasics.dtx, b=fimain.dtx, c=ficonv.dtx, d=filtfam.dtx, e=fimapgen.dtx

F

G

File Key: a=fibasics.dtx, b=fimain.dtx, c=ficonv.dtx, d=filtfam.dtx, e=fimapgen.dtx

File Key: a=fibasics.dtx, b=fimain.dtx, c=ficonv.dtx, d=filtfam.dtx, e=fimapgen.dtx

N

O

P

Q

R

File Key: a=fibasics.dtx, b=fimain.dtx, c=ficonv.dtx, d=filtfam.dtx, e=fimapgen.dtx

S

T

U

File Key: a=fibasics.dtx, b=fimain.dtx, c=ficonv.dtx, d=filtfam.dtx, e=fimapgen.dtx

Change History

v 1.6

General: AFM commands fixed, to get fontdimens comparable to EC fonts. (Thierry Bouche) 94

\fromany: Search order changed to PL before AFM. (SPQR) The code wasn't in \fromany back then, though. 107

v 1.800

General:

\textcompfamily integrated into \latinfamily. (UV) 120

fontinst.sty and fontdoc.sty now generated from fontinst.dtx. (UV) . 17

v 1.900

General:

VPL-specific properties added to those which are ignored by \pltomtx. (LH) 100

New method of storing slot assignments. (LH) 71

Added compability code for the old interface for boundary ligatures and kerns. (LH) 71

Method of computing italic corrections changed to using an integer expression. (LH) 96

fontinst.dtx split into several source files. (LH) 17

Replaced \relax by \x_relax. /LH 23

Replaced \resetint by \x_resetint. (LH) 39

Replaced \setint by \x_setint. (LH) 39

Replaced \setstr by \x_setstr. (LH) 39

\afm_length: Macro added, other macros modified to use it. (LH) 93

\encoding: Changed group in fontdoc to \begingroup type. (LH) 48

File Key: a=fibasics.dtx, b=fimain.dtx, c=ficonv.dtx, d=filtfam.dtx, e=fimapgen.dtx

Internal notes

A Typographic treatment

- I have been following *The LaTeX Companion* in that I have set all names of packages and the like—fontinst, fontdoc, trig, doc, and docstrip (I might have forgotten some)—in sans serif type. Actually, I have defined a command `\package` in `fisource.sty` (or is that fisource?) which does this, so if we decide on some other formating, we can just change that. /LH

- It seems to me that there should indeed be some space between the 'v' and the digits in a version number when it is typeset, but I also think a full space is too much, so I have been using thin spaces. These are unbreakable and under LaTeX you can simply use `\,` to get one (while you are not in math mode, that is). /LH

A

- While I went through (some of) the code, I came across a few inconsistencies. **I have marked them out like this**—some boldfaced text in a paragraph and a large A in the margin. I defined a command `\ambiguity` for doing this. /LH

?

- There is also a similar command `\question` which is intended for situations where there isn't really an error, but somethings seems like it should be taken care of in some other way. **The `\question` command puts a question mark in the margin.**

Not in doc: `\foo`

- I also noticed that there are several fontinst commands, not all of which are new, which are not defined in fontdoc. To mark out such things, I have written things like the one shown in the margin by this paragraph. I defined a command `\missing` for doing this. /LH

B Planning topics

This section lists items in the larger design of fontinst which need to be resolved in one way or another. Debates about these items on the fontinst mailing list are welcome.

B.1 Reorganisation of the source

At the moment, all the ideas suggested have been realized.

B.2 Files

Which files should be considered temporary and placed in the location specified with `\tempfileprefix`? Should files be explicitly be looked for at this location or should it be assumed that fontinst users include that location in their TeX input file search path? In the former case, *which* files should be looked for in that location? Should files be looked for in the temporary location before they are looked for without a specified location, or should it be the other way round?

C Contributors

[The fontinst source has been pretty inconsistent in how people are credited for what they have done—some appear only as acronyms while others appear as rather striking e-mail addresses—so I thought it best that this is shaped up a bit. My suggestion is that we use names or acronyms in the source and move everything else here. I also thought it could be interesting with a short description of what each person has done and is doing, so I have started ever so slightly on something along those lines. Feel free to add things! /LH]

The following people have contributed substantial amounts of code or documentation to fontinst. They are listed in strict alphabetical order.

Thierry Bouche Thierry saw to that the `T1` encoded fonts got font dimensions comparable to the `ec` fonts. Thierry is also the author of several papers (published in the *Cahiers GUTenberg* and *TUGboat*) which deal with non-trivial applications of fontinst, such as creating metrics for Adobe Minion Multiple Master fonts and developing a corresponding math font setup.

E-mail: `Thierry.Bouche@ujf-grenoble.fr`

Lars Hellström (LH) Lars is responsible for most of the things in v 1.9 that were not there in v 1.8.

Lars is currently a member of the fontinst maintenance team. He is also a doctorate student in mathematics at Umeå University.

Alan Jeffrey (ASAJ) Alan is the original author of fontinst. He is not currently on the development team but he is a frequent writer on the fontinst mailing list, which he manually manages subscriptions to.

E-mail (fontinst mailing list subscriptions):
`fontinst-request@cogs.susx.ac.uk`.

Constantin Kahn (CK) Constantin is one (Sebastian is the other) of the original coauthors of the current `\latinfamily` command.[7]

Rowland McDonnell Rowland rewrote Alan's old v 1.5 documentation for fontinst and updated it for v 1.8.

Sebastian Rahtz (SPQR) Sebastian is one (Constantin is the other) of the original coauthors of the current `\latinfamily` command. He has also contributed numerous ETX files and made the "unofficial" v 1.6 and v 1.7, which included the first fontinst support for making `TS1` encoded files.

Sebastian is currently a member of the fontinst maintenance team. Other TEX-related occupations include the `fonts/psfonts/` subtree of CTAN, the TEX-live CD collection, ...

Ulrik Vieth (UV) Ulrik converted `fontinst.sty` to doc/docstrip format, reunited Alan's v 1.511 and Sebastian's v 1.7, and made the first official release of fontinst (v 1.8) for more than two years.

Ulrik is currently a member of the fontinst maintenance team. He is also involved in the Joint TUG/LATEX 3 Project Working Group on extended math font encodings.

[7] Am I right about this? It's a bit before my time, I've just read `CHANGES`. /LH

...

Anyone else?

D To do

This section is based on the `TODO` file from fontinst v 1.504, but a couple of new entries have been added and some have been equipped with comments.

D.1 Things to do in the "near" future

- Update documentation. (Lars, knowing he isn't saying anything new)
- Update `fontdoc.sty` and co. for LaTeX 2_ε. (Alan)

  Some of this was done in v 1.8 when a proper package interface was added and some has been done in v 1.909, but there is still a lot to do. /LH
- `latin.mtx` fakes composite SC glyphs from the composite glyph, eg.

  ```
  \setleftrightkerning{Aacutesmall}{Aacute}{900},
  ```

  rather than from the SC non-composite, eg.

  ```
  \setleftrightkerning{Aacutesmall}{Asmall}{1000}
  ```

  This may cause problems with SC fonts with explicit SC–C kerns, eg.

  ```
  \setkern{V}{Asmall}{100}
  ```

  This needs to be thought about. Pointed out by Hilmar Schlegel.
- Rewrite the entire substitution mechanism from scratch! The main problem with the current mechanism is that it only allows one substitution per shape and one per series. One cannot substitute the `it` shape for both the `sl` and the `ui` shapes since each new `\substitute`(`noisy`|`silent`) with `it` in the ⟨*from*⟩ argument will overwrite the setting made by the previous;

  ```
  \substitutenoisy{ui}{it}
  \substitutesilent{sl}{it}
  ```

  is effectively the same as

  ```
  \substitutesilent{sl}{it}
  ```

  Another big problem is that it isn't well defined what the substituion mechanism should do. (Lars)

  I've got a sketch for a new substitution mechanism, but I'm not at all sure it will make it into any v 1.9xx. /LH
- Consider writing an `\installfontas` command which doesn't generate a (V)PL but makes an entry in the FD file. Typical usage: Install an `sc` shape of a `TS1` font family by using the font made for the `n` shape. (Lars)

- Use the `StdVW` property from AFM 4.0 files instead of the width of *I* for standard stem fontdimen. Pointed out by Hilmar Schlegel.
- Consider removing the following unreliable fakes from `textcomp.mtx`:

  `asciiacutedbl`, `asciigravedbl`, `bardbl`, `openbracketleft`, and `openbracketright`

  and consider adding fakes for the following unavailable glyphs:

  `dollaroldstyle` (use `dollar`), `centoldstyle` (use `cent`), `lira` (use `sterling`), and `pilcrow` (use `paragraph`)

  These suggestions are of course open for debate. (Ulrik)

D.2 Things that probably won't be done in the near future

- Find a way to automatically generate math fonts. (Alan)

  I doubt that this will ever be possible to do automatically. mathptm and mathptmx are already hackish enough, not to mention the prototype implementations for new math font encodings. /UV
- Investigate using Alternate sets. (Alan)

  AFAIK, Thierry Bouche has done some work with alternate sets for AGaramond, ACaslon and Minion, but this implies a lot of manual work to compose the proper calls to fontinst. /UV
- Create LaTeX packages? In fact, rethink the whole package interface ... (Alan)

  Sebastian's Perl front-end to fontinst does create trivial LaTeX packages automatically for the CTAN fonts. /UV
- Worry about excessive kern tables in `T1` fonts. (Alan)

  I think the best way to get at this would be to write a program that can optimize (for size) kern tables by making use of the `SKIP` instruction. There's often room for quite a lot of compression. Such a program would however have to be written in some compiling language, otherwise it wouldn't be fast enough. /LH
- Consider making `\set(left|right)kerning` parameterized by the size of the other glyph, eg so that faking `<Asmall><T>` can be different from `<Aacutesmall><T>`. Suggested by Hilmar Schlegel.

D.3 Things that have been done

- Update documentation (comments from Karl and Damian).

  Rowland has meritoriously done the update requested here. On the other hand, it needs to be updated again, since many new features have been added. /LH

- Investigate using raw SC fonts. (Alan)

  This problem comes up on the mailing list from time to time. The problem is that fontinst's `\latinfamily` command is geared towards fonts that come with standard and expert font sets (as with Adobe and Monotype fonts) rather than fonts that come with standard and small caps (as with Linotype fonts). I'm afraid it would be to complicated to handle both cases in the same `\latinfamily` procedure. /UV

- Investigate problems with duplicate kerns appearing in VPL files (Hilmar Schlegel).

  The problem has been investigated and is solved with v 1.9. /LH

- Find out why, if you have a font with both medium and light variants but no italic, you get `m/it` $\mapsto$ `l/it` $\mapsto$ `l/sl` rather than `m/it` $\mapsto$ `m/sl`. (Sebastian).

  Because for every shape that fontinst is allowed to use a given font for, it will perform all possible series substitutions. If the `\installfont` for `l/sl` came before the `\installfont` for `m/sl`, then substitutions will be as described above. See also item about the substitution mechanism. /LH

- AFM files can contain real units, not just integers. (Gintautas Grigelionis).

  This is fixed with v 1.9. /LH

- Make `fontdoc.sty` and co. use PS fonts. (Alan)

  I think `\useexamplefont` and friends (introduced with v 1.8) pretty much do what was intended here. /LH

- `latin.mtx` uses fontinst rather than fontdoc! (Rob Hutchings).

- Allow `.vpl` files to be read as `.pl` files. (Constantin)

- Richard Walker reports that if you say

  ```
  \latinfamily{mbvx}{}
  \latinfamily{mbv9}{}
  ```

  then the 2nd run doesn't use old-style digits, because the 1st run has already defined `\digit`. If so, this is because `\latinfamily` is missing a bracing level. I should investigate.

  I think it is best to do oldstyle and non-oldstyle variants in two separate fontinst runs, i.e. don't use `\latinfamily` more than once in a single run. (IIRC, grouping doesn't work propoerly since every single font closes and reopens a `\begingroup`-`\endgroup` pair to encapsulate the kerning info[8] or something like that.) /UV

D.4 Other notes

Alan's `TODO` file also contains some items regarding mathptm, but that seems to have migrated out of fontinst (if it ever really was a part). The problem is that mathptm cannot be changed any more for the sake of checksum consistency and backwards compatibility. A new variant called mathptmx tries to do a little better, but there may still be room for improvements.

[8] And glyph metrics info, and glyph mapcommands info ... /LH

E Efficiency

This section records the results of some (rather simple) test runs that have been made to test the efficiency of fontinst, primarily to see how changes in the implementation affect efficiency by comparing the time and space used by different fontinst versions to complete the same task.

E.1 Alan Jeffrey's tests

I compared the version where you try to keep ints as `\mathchardef`s with the version where you don't bother, and for a sample font without `\mathchardef`s I got:

> 114673 words of memory out of 150001
> Time elapsed: 135.0 seconds

and with, I got:

> 114050 words of memory out of 150001
> Time elapsed: 134.5 seconds

so I've saved a little memory and time. Not brilliant, but I may as well keep it in.

Where possible, we avoid re-scaling kerns, which saves a bit of time and memory. With a sample font, the version where we didn't avoid re-scaling used:

> 114050 words of memory out of 150001
> Time elapsed: 134.5 seconds

whereas the version where we do avoid it used:

> 113786 words of memory out of 150001
> Time elapsed: 124.9 seconds

We keep the names of the glyphs to kern with as `\l-`⟨*name*⟩ and `\r-`⟨*name*⟩ to save on token space, and this got the resources used down to:

> 88574 words of memory out of 150001
> Time elapsed: 106.1 seconds

Keeping track of the kern amounts as `\`⟨*amount*⟩ got the resources used down to:

> 75424 words of memory out of 150001
> Time elapsed: 97.2 seconds

Mind you, I then added all the `\transformfont` stuff, and it went back to:

> 77079 words of memory out of 150001
> Time elapsed: 97.7 seconds

E.2 Current tests

The setup for this test is that TEX is run on a `.tex` file consisting of the following commands.

```
\batchmode
\input fontinst.sty
\latinfamily{pad}{}
\tracingstats=1\bye
```

All the font metrics are generated from the AFM files; temporary MTX, PL, and VPL files are deleted between test runs. The format used was generated by typing `\dump` at iniTEX's `**` prompt; thus there is no overhead from a typesetting format in the space requirements.

[It appears the exact data from these tests will have to wait, since I haven't been able to get access to any computer with reliable process timing. For the next release perhaps ... /LH]

References

[1] Donald E. Knuth, Duane Bibby (illustrations): *The TEXbook*, Addison Wesley, 1991; volume A of *Computers and typesetting*.

[2] Karl Berry: *Fontname*, CTAN:`info/fontname/fontname.texi` (and most of the other files in that directory as well).

Some other things which definitely ought to be in the bibliography, but for which I couldn't find references just when I was at it are:

The LaTeX companion
The LaTeX graphics companion
LaTeX 2_ε font selection (`fntguide.tex`).