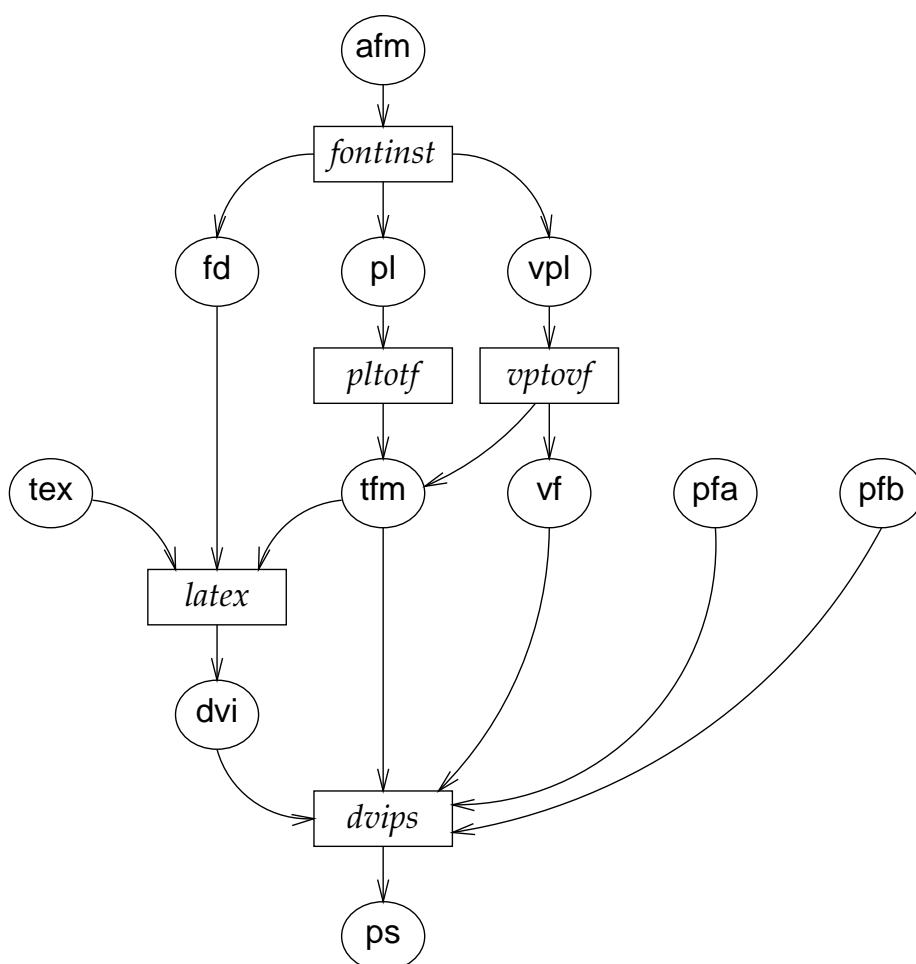


fontinst

Font installation software for T_EX



Alan Jeffrey Rowland McDonnell
Lars Hellström
fontinst v1.9 · July 2004

This manual is currently being rewritten, and may therefore be a bit disorganised. For authoritative information on command syntaxes and the like, see the Literate Programming sources for fontinst. The main file there is `fisource.tex`.

This manual describes the `fontinst` software for converting fonts from Adobe Font Metric format to forms readable by \TeX . This manual should be distributed with the `fontinst` software, which is available by anonymous FTP from `ftp://ftp.tex.ac.uk/tex-archive/fonts/utilities/fontinst`, and on the various CD-ROMs containing material from the CTAN archives. Please do not contact the author directly for copies.

If you would like to report a bug with `fontinst`, please mail `fontinst@tug.org`. The mail will be sent to the `fontinst` mailing list. If you would like to be on the `fontinst` mailing list, see <http://tug.org/mailman/listinfo/fontinst>.

Contents

1	Defining terms	4
1.1	What's a font?	4
1.2	What does fontinst do?	7
1.3	What do you do with fontinst?	8
2	Fontmaking commands	9
2.1	Install commands	9
2.2	Transformation commands	13
2.3	Miscellaneous settings	14
2.4	Other	16
3	Mapmaking commands	16
4	General commands	17
4.1	Variables	17
4.2	General commands	18
4.3	Integer expressions	19
4.4	Other	20
5	Encoding files	21
5.1	Encoding commands	22
5.2	Slot commands	22
5.3	Other	23
6	Metric files	24
6.1	Metric commands	24
6.2	Glyph commands	26
6.3	Other	29
7	fontdoc commands	29
8	fontinst variables	29
9	Customisation	37

10	Notes on features new with v 1.9	37
10.1	Metric packages	37
10.2	Word boundary ligatures and kerns	39
10.3	Changing the names of glyphs	40
10.4	Making map file fragments	42
10.5	Tuning accent positions—an application of loops	45
10.6	Font installation commands	46
10.7	Bounding boxes	49
11	Miscellaneous notes	50
11.1	On verbatim, typewriter, and monowidth fonts	50

1 Defining terms

This is rather a large and perhaps tedious section. You might be tempted to skip it so you can get to some more direct information on how to use fontinst. That's fine if you understand everything about how T_EX handles fonts. If not, I suggest you at least skim through this section.

1.1 What's a font?

Once upon a time, this question was easily answered: a font is a set of type in one size, style, etc. There used to be no ambiguity, because a font was a collection of chunks of type metal kept in a drawer, one drawer for each font.

These days, with digital typesetting, things are more complicated. What a font 'is' isn't easy to pin down. A typical use of a PostScript font with L^AT_EX might use these elements:

- Type 1 printer font file
- Bitmap screen font file
- Adobe font metric file (`afm` file)
- T_EX font metric file (`tfm` file)
- Virtual font file (`vf` file)
- font definition file (`fd` file)

Looked at from a particular point of view, each of these files 'is' the font. So what's going on?

1.1.1 Type 1 printer font files

These files contain the information needed by your printer to draw the shapes of all the characters in a font. They're typically files with a `pfa` or `pfb` extension; on Macs they're usually in files of type 'LWFN' which usually have icons that look like a laser printer. The information in all these files is basically the same: the only difference is in its representation. `pfa` stands for 'printer font ASCII', while `pfb` stands for 'printer font binary'. That is, `pfa` files contain plain text information, while `pfb` files contain the same information encoded as machine-readable binary data.

If you have Adobe Type Manager (ATM) installed on your computer, ATM will use these files to draw an accurate representation of the letters on the screen of your computer when you are previewing a T_EX document.

Printer font files are not used directly by T_EX at all – T_EX just prepares a `dvi` file that refers to the fonts by name and the characters by number: T_EX knows nothing about the shapes involved. The DVI driver uses the printer font files when you ask it to print the `dvi` file. This means that you can produce a `dvi` file which uses, say, Palatino, even if you do not have the Type 1 printer font

file for this font on your computer. You will need to find a computer that does have Palatino before you can print it or preview it, though.

Pdf \TeX is different from \TeX in this respect; since pdf \TeX integrates most of the functionality of a DVI driver, it may be unable to generate working pdf output if the some Type 1 printer font file is not available.

1.1.2 Bitmap screen font files

These files contain a low-resolution bitmap for drawing a representation of the font on the screen of your computer if ATM is not installed. In the \TeX world, these files are only used for screen previews by the DVI driver. They are kept in font suitcase files on Macintoshes.

1.1.3 Adobe font metric files (afm files)

These files are text files which contain information about the size of each character in a font, kerning and ligature information, and so on. They can't be used by \TeX directly, but the information they contain is essential if you want to use a font with \TeX . Fontinst can create from an afm file the necessary tfm and vf files so you can use a font with \TeX . Once you have created all the files you need to use a font with \TeX , you can remove the corresponding afm files from your computer unless you have other software that needs them.

The job of turning an afm file into a set of tfm and vf files is one of the main uses for fontinst. Most of this document is concerned with this process, so don't worry if it seems a bit vague at the moment.

1.1.4 \TeX font metric files (tfm files)

These are binary data files in a format designed for use by \TeX which contain (more-or-less) the same information as afm files: the size of each character in a font (font metric data), kerning, and ligature information.

When you select a font in \TeX , you are telling \TeX to typeset using a particular tfm file; from \TeX 's point of view, a tfm file (and nothing else) *is* a font. \TeX itself doesn't see printer font files, screen bitmaps, pk files, vf files, or anything else to do with fonts: only tfm files.

\TeX uses these tfm files to decide where to put characters when typesetting. From \TeX 's point of view, tfm files are fonts, even though they contain no information about the shape of letters, and are not used by anything except \TeX – once you have produced a dvi file, you don't need the tfm files to print it out. (This is a slight lie: dvips can read tfm files corresponding to PostScript and TrueType fonts so it can modify the metrics slightly to improve the letter-spacing at your chosen output resolution. This is an optional minor tweak and not an essential part of the output process.)

1.1.5 Property list files (pl files)

pl files are human-readable text files which contain all the font metric, kerning, ligature, and other information needed to create a tfm file. You can convert between the two file formats using tftopl and pltotf.

1.1.6 Virtual font files (vf files)

These are binary data files in a format designed for use by T_EX dvi drivers. Their main purpose in life is to let you use fonts in different encodings to the standard T_EX encodings. These files are used by dvi driver software only.

They are used only by dvi drivers to work out what it should *really* print when you ask for a particular character. Technically they are like subroutine libraries for dvi drivers, with one subroutine for each character in the virtual font: when the dvi driver sees a dvi command to set a character from a virtual font, it will execute a sequence of dvi commands (the “mapcommands property” of this character) that it reads in the vf file. You need not worry about the details of this, as fontinst deals with them for you. Creating and using virtual fonts is what this document is about, so don’t worry if this doesn’t make sense yet. (After all, how much do you need to know about the inner workings of dvi files to typeset and print T_EX documents?)

Each vf file has a tfm file with the same name. To use a virtual font, you select the tfm file as the font to use in your document. When the dvi driver comes across this tfm file in the dvi file, it looks up the vf file and uses that to decide what to do.

1.1.7 Virtual property list files (vp1 files)

vp1 files are human-readable text files which contain all the font metric, kerning, mapping, and other information needed to create a vf and tfm pair.

vptovf will create a vf/tfm pair from a vp1 file. vftovp will create a vp1 from a vf/tfm pair. vftovp also needs to be able to read all the tfm files that are referred to by a vf to recreate the vp1 – it looks at the checksums to verify that everything’s okay.

1.1.8 Font definition files (fd files)

These are files containing commands to tell L^AT_EX which tfm files to associate with a request for a font using L^AT_EX’s font selection commands.

For example, here is a small and edited part of the fd file supplied with PSNFSS to allow you to use the Adobe Times font in T1 encoding:

```
\ProvidesFile{t1ptm.fd}
[1997/02/11 Fontinst v1.6 font definitions for T1/ptm.]
```

```

\DeclareFontFamily{T1}{ptm}{}

\DeclareFontShape{T1}{ptm}{m}{n} {<-> ptmr8t}{}
\DeclareFontShape{T1}{ptm}{m}{it}{<-> ptmri8t}{}
...
\DeclareFontShape{T1}{ptm}{b}{n} {<-> ptmb8t}{}
\DeclareFontShape{T1}{ptm}{b}{it}{<-> ptmbi8t}{}
...

```

What this means is: when you use \LaTeX to select the font family `ptm` in `T1` encoding in the medium series (`m`) and normal shape (`n`), \TeX uses the font `ptmr8t.tfm`. Similarly, if you select bold italic, \TeX uses `ptmbi8t.tfm`.

\LaTeX works out which `fd` file to load based on the current encoding and font family selected. If you've selected `T1` encoded `ptm` like this:

```
\fontencoding{T1}\fontfamily{ptm}\selectfont
```

\LaTeX loads the file `t1ptm.fd` (if it doesn't exist, you're in trouble). As you can see above, this file contains information so that \LaTeX knows which `tfm` file to use. So if you ask for, say, `T1/ptm/b/it` (`T1` encoded Times-Roman, bold series, italic shape), you get the font `ptmbi8t`.

You can find more about `fd` files and \LaTeX 's font selection commands at CTAN: <ftp://ftp.tex.ac.uk/tex-archive/macros/latex/base/fntguide.tex> and <ftp://ftp.tex.ac.uk/tex-archive/info/simple-nfss.tex> are both useful.

1.2 What does fontinst do?

`fontinst` creates `vp1` and `p1` files from `afm` or `p1` files to map any glyph or combination of glyphs in the original font files to any slot in the output font file. There, isn't that better? Off you go now...

If you're still confused, I'll explain a few things.

Glyph A glyph is an image, often associated with one or several characters. Some examples of glyphs are: 'A', 'A', 'A', 'B', 'F', 'f', 'fi', '~'. Fonts are collections of glyphs. `Fontinst` refers to glyphs by name.

Slot This is jargon for 'a numbered position in a font'. (What is important is the number, and that this number refers to a position in a font, but which font is usually irrelevant.)

Character The modern definition is that a character is the smallest component of written language that has semantic value. Speaking of a character, one refers to the abstract meaning and/or shape, rather than a specific shape.

Since fonts have often contained a unique glyph for each character and each usable glyph has been assigned a particular slot, it is not uncommon (in particular in older terminology) to see the three mixed up, usually so that one says 'character' where one of the other two would have been more correct. The \TeX -related font file formats is no exception, as you may see examples of elsewhere in this document.

Encoding There are really two different encoding concepts that one encounters when using fontinst. The differences are not great, and an encoding of one kind often corresponds to an encoding of the other kind, but it is not merely a matter of translation.

A *LaTeX encoding* is a mapping from characters (or more formally LaTeX Internal Character Representations) to slots. In the OT1 encoding, ‘ø’ (or more technically ‘\o’) maps to slot 28, whereas in the T1 encoding it maps to slot 248. This kind of encoding affects what TeX is doing; dvi drivers are not involved.

A *font encoding* (or *encoding vector*) is a mapping from slots to glyph names. This is the kind of encoding that fontinst primarily deals with, and also the kind of encoding that dvi drivers make use of. ot1.etx associates slot 28 with ‘oslash’, whereas t1.etx and EC.enc (one of several to T1 corresponding encoding vectors that come with dvips) associates slot 28 with ‘fi’.

LaTeX encodings occur in fontinst only as names and only in relation to fd files. It is unlikely that you will need to create one of your own. The mappings defined by font encodings are on the other hand of great importance and etx files are used to direct the generation of virtual fonts. Advanced fontinst users may well find that they need to create new font encodings to achieve their goals.

The thing is that the average PostScript font comes in Adobe standard encoding, which, for example, has the glyph dotless i ‘ı’ in slot 245. But TeX T1 encoding expects the glyph o dieresis ‘ö’ in that slot, and wants dotless i in slot 25. So if you tried to use a raw PostScript font with TeX, any time you tried to get a ‘ö’, you’d get a ‘ı’; and every time you tried to get a ‘ı’, you’d get a blank, because Adobe standard encoding says that slot 25 is empty. The process of dealing with this problem is called ‘re-encoding’, and is what fontinst helps with.

This might not make much sense yet; the best thing to do is relax. There’s a lot of things that need to be dealt with when you’re setting up LaTeX to use a new font, so you can expect to be a bit confused until you’ve done it a few times.

1.3 What do you do with fontinst?

If you’re using fontinst, the usual steps you need to take to use an ordinary PostScript text font with LaTeX are these:

1. Give the afm files an appropriate name.
2. Use fontinst to produce an 8r encoded p1 files from these afm files.
3. Use fontinst to create T1 and OT1 encoded p1 and vp1 files from the 8r encoded p1 files (this procedure will also create suitable fd files).
4. Use pltotf to turn each p1 file into a tfm file.
5. Use vptovf to turn each vp1 file into a pair of vf and tfm files.
6. Move the tfm, vf, and fd files into the appropriate directories so LaTeX can see them.

7. Tell your DVI driver about the new font (edit dvips's `psfonts.map` file, or OzTeX's Default configuration file).
8. Perhaps write a package file to make selecting the new font a little easier.
9. Test it.

2 Fontmaking commands

There are three main types of files that you may write to control what fontinst does: *command files* (usually with suffix `.tex`), *encoding definition files* (suffix `.etx`), and *metric files* (suffix `.mtx`). Command files directly tell fontinst to do things, whereas the purpose of an encoding or metric file is more to store data, but all three file types are technically sequences of TeX commands that fontinst execute when reading the file. Normal TeX syntax rules apply in all three file types, although a few commands may behave in unfamiliar ways.

Within the command file category, it is possible to discern certain subcategories. Most command files are written for one particular task, but some are common pieces that have been factored out from larger command files and are merely meant to be `\input` where appropriate. (`csc2x.tex` in the fontinst distribution is an example of this latter kind.) One may also distinguish between command files that are made for use with `fontinst.sty` command definitions and command files that are made for use with `fontinstmsc.sty` command definitions. This section documents the commands that are particular to the former category, whereas the next section documents commands that are particular to the latter.

2.1 Install commands

The core fontmaking takes place within a block of “install commands”. (This name is a bit unfortunate since nothing is actually installed; rather some files that need to be installed are generated.) Such blocks have the structure

```
\installfonts
<install commands>
\endinstallfonts
```

The `<install commands>` describe the fonts, glyphs and encodings used to build fonts, whereas the purpose of the delimiting `\installfonts` and `\endinstallfonts` are rather to organise the writing of fd files.

```
\installfonts
\endinstallfonts
```

At `\installfonts`, fontinst's internal list of fd files to generate are cleared. At `\endinstallfonts`, fd files are written for those combinations of encoding and font family that appeared in the `<install commands>`.

Note for hackers. `\installfonts`, `\endinstallfonts`, and the individual install commands between them also cooperate in a rather complicated grouping scheme to cache glyphbases. This may interfere with non-fontinst commands in the *install commands*. If for example an assignment to some `\tracing...` parameter here does not seem to have any effect, try making the assignment `\global`.

The most important *install command* is

```
\installfont{font-name}{metrics-list}{etx-list}
{encoding}{family}{series}{shape}{size}
```

This produces a T_EX virtual font called *font-name*. The *metrics-list* and the *etx-list* determine this font, whereas the other arguments specify how the fd file will declare it for L^AT_EX. The *encoding*, *family*, *series*, and *shape* are precisely the NFSS parameters. The *size* is either a shorthand declared by `\declaresize` (see below), or is an fd size specification.

Like most fontinst lists, the elements in the *metrics-list* and *etx-list* are separated by commas (so-called comma-separated lists). In their simplest form, the elements of these lists are file names (minus suffixes): `mtx` files in the *metrics-list* and `etx` files in the *etx-list*. First the `mtx` files are processed to build up a glyphbase, i.e., store definitions of glyphs and their metric properties in memory, and then the `etx` files are processed (several times) to select a set of glyphs and write the corresponding information to a `vp1` file.

For example, to install the T1-encoded Times Roman font (using `t1.etx` and `latin.mtx`), you say:

```
\installfont{ptmr8t}{ptmr8r,latin}{t1}
{T1}{ptm}{m}{n}{}
```

To install a OT1-encoded Times Roman font with a scaled version of Symbol for the Greek letters, you say:

```
\installfont{zptmrsy}{ptmr8r,psyr scaled 1100,latin}{ot1}
{OT1}{ptm}{m}{n}{}
```

As the second example indicates, there is more to the list items than just file names. In the case of an metrics list item, the syntax permits the two forms

```
<filename><optional modifiers>
\metrics <metric commands>
```

where an *optional modifier* is one of

```
␣scaled␣<rawscale factor>
␣suffix␣<glyph name suffix>
␣encoding␣<etx>
␣option␣<string>
```

A list item may contain several such modifiers, but most commonly it does not contain any. The *metric commands* are explicit metric commands, as described

New feature
v1.923

in Section 6; this latter feature is meant for minor adjustments that you don't want to bother creating a separate `mtx` file for.

The `<filename>` above primarily refers to a file `<filename>.mtx`, but that need not always exist before executing the above command. If there exists a `pl`, `afm`, or `vp1` file with the right name then that is first converted to a corresponding `mtx` file. However, a special case occurs if there is an encoding modifier: this forces conversion of a `pl` or `vp1` file even if an `mtx` file exists, and also forces using the specified `etx` file when assigning glyph names to the slots of that file. Normally the choice of `etx` file for such conversions to `mtx` is based on `\declareencoding` declarations.

The `scaled` modifier sets the `rawscale` variable for the processing of that file. This has the effect of scaling all raw glyphs from that file to `<rawscale factor>` per milles of their previous size. The `suffix` modifier causes the `<glyph name suffix>` to be implicitly appended to all glyphs defined by this file. The `option` modifier adds the `<string>` to the list of "options" for this file. The `\ifoption` command can be used in the file to test whether a particular string has been supplied as an option.

Note for hackers. In general, `fontinst` commands process comma-separated list arguments by first splitting at commas and then fully expanding each item, but this `<metrics-list>` argument is an exception. This is first fully expanded (`\edef`) and then split into items. The difference is that a macro used in this `<metrics-list>` argument can expand to several list items, whereas a macro used in an ordinary comma-separated list argument can only expand to (part of) a single list item.

The `\metrics` list items do however constitute an exception within this exception. These list items are in their entirety protected from the initial full expansion, so you don't have to worry about peculiar fragility errors there.

The elements in the `<etx-list>` have fewer variants, but there is still a general syntax

`<filename><optional modifiers>`

The only `<optional modifier>` permitted is however

`\mtxasetx`

and that is probably only relevant for use with `\installrawfont` (see below).

`\installfontas{<font-name>}{
 {<encoding>}{<family>}{<series>}{<shape>}{<size>}}`

This `install` command adds an `fd` entry for the `<font-name>`, but it doesn't actually generate that font. Usually that font was generated by a previous `\installfont`, and this is used to create additional entries for the font.

New feature
v1.912

<code>\installrawfont{<font-name>}{<metrics-list>}{<etx-list>}{<encoding>}{<family>}{<series>}{<shape>}{<size>}</code>
--

This is similar to `\installfont` except that it produces a \TeX raw font as p1 file rather than a virtual font. Often a p1 font with the specified name will already exist when this command is called, and that will then be overwritten. These two p1 files will typically be somewhat different. The normal reason for using this command is that one wishes to “refine” the metrics of a font that was generated by transformation commands.

For example, to install an 8r-encoded Times Roman raw font (using `8r.etx` and `8r.mtx`), you say:

```
\installrawfont{ptmr8r}{ptmr8r,8r}{8r}
{8r}{ptm}{m}{n}{}
```

The effect of a

<code><filename>_mtxasetx</code>
--

in the `<etx-list>` is not that `<filename>.etx` is read, but that `<filename>.mtx` is read. The interpretation of the commands in this file is however not the customary, and almost the only thing paid attention to is the correspondence between glyph names and slot numbers that is provided by the `\setrawglyph` and `\setscaledrawglyph` commands; this correspondence is treated as if it was given by `\setslot ... \endsetslot` commands in an `etx` file. This is however only guaranteed to work with transformable metric files.

New feature
v1.923

The purpose of this feature is to simplify installation of fonts with very special encodings, such as “Dingbat” or “Pi” fonts. Instead of creating an `etx` file, which would probably only be useful with that particular font, one can make use of the fact that the interesting information is anyway available in the `mtx` file. To install Zapf Dingbats in their default encoding, one can thus say

```
\installrawfont{pzdr}{pzdr}{pzdr mtxasetx}
{U}{pzd}{m}{n}{}
```

Unlike the case with `\installfont`, which actually creates a real (although virtual) font, `\installrawfont` can only create the metrics for a font. The dvi driver will require some other kind of implementation of this font, usually an entry in some map file (e.g. `psfonts.map`, in the case of dvips) that links the \TeX font name to e.g. a PostScript font name and file. (Many dvi drivers are configured in such a way that they, without such a map file entry, will call Metafont with the font name and thereby raise a sequence of error messages about a `.mf` that doesn’t exist. These results are often rather confusing.)

<code>\installfamily{<encoding>}{<family>}{<fd-commands>}</code>
--

This tells fontinst to write an `fd` file for the given combination of encoding and family, and clears the internal list of entries to put in that file. `\installfamily` commands usually come first in each block of `<install commands>`.

For example, to produce a L^AT_EX family with the Cork-encoded Times family, you say:

```
\installfamily{T1}{ptm}{}
```

The *⟨fd-commands⟩* are executed every time a font in that family is loaded, for example to stop the Courier font from being hyphenated you say:

```
\installfamily{T1}{pcr}{\hyphenchar\font=-1}
```

In more recent versions of fontinst, the `\installfamily` command is only necessary if you want the *⟨fd-commands⟩* argument to be nonempty, but it doesn't hurt to make it explicit.

2.2 Transformation commands

```
\transformfont{⟨font-name⟩}{⟨transformed font⟩}
```

This makes a raw transformed font, for example expanded, slanted, condensed or re-encoded. *It is the responsibility of the device driver to implement this transform.* Each `\transformfont` command writes out an `mtx` file and a raw `p1` file for *⟨font-name⟩*.

A *⟨transformed font⟩* is given by the following commands:

```
\fromafm{⟨afm⟩}
\fromany{⟨whatever⟩}
\frompl{⟨pl⟩}
\fromplgivenetx{⟨pl⟩}{⟨etx⟩}
\frommtx{⟨mtx⟩}
```

This reads the metrics of a font which is about to be transformed from an external file. `\fromafm`, `\frompl`, and `\fromplgivenetx` write out an `mtx` file corresponding to the `afm` or `p1` file. In addition, `\fromafm` also writes out a raw `p1` file, containing just the glyph metrics but no kerning information. `\fromplgivenetx` permits specifying which encoding file to use when associating glyph names to slots, whereas `\frompl` tries to guess this from the `CODINGSCHEME` property of the `p1` file. `\fromany` looks for a file in any of the formats (in the order `mtx`, `p1`, `afm`) and behaves as the first `\from...` for which it found a file.

A *⟨transformed font⟩* may also be one of the following:

```
\scalefont{⟨integer expression⟩}{⟨transformed font⟩}
\xscalefont{⟨integer expression⟩}{⟨transformed font⟩}
\yscalefont{⟨integer expression⟩}{⟨transformed font⟩}
\slantfont{⟨integer expression⟩}{⟨transformed font⟩}
```

This applies a geometric transformation to the font metrics of *⟨transformed font⟩*. The scale factor or slant factor are given in 1000 units to the design size. Typical examples are 167 for slanted fonts or 850 for condensed fonts.

The final case of a $\langle transformed font \rangle$ is:

`\reencodefont{ $\langle etx \rangle$ }{ $\langle transformed font \rangle$ }`

This rearranges the encoding vector of $\langle transformed font \rangle$ to match the encoding given by the `etx` file.

For example, to create an oblique, 8r-encoded version of Adobe Times called `ptmro8r` you say:

```
\transformfont{ptmro8r}{\reencodefont{8r}{\slantfont{167}{\fromafm{ptmr8a}}}}
```

This will create `ptmr8a.mtx`, `ptmr8a.pl`, `ptmro8r.mtx` and `ptmro8r.pl`, which can then be used as raw fonts in `\installfont` commands. The same transformation can also be achieved in two steps:

```
\transformfont{ptmr8r}{\reencodefont{8r}{\fromafm{ptmr8a}}}  
\transformfont{ptmro8r}{\slantfont{167}{\frommtx{ptmr8r}}}
```

This will create `ptmr8a.mtx`, `ptmr8a.pl`, `ptmr8r.mtx`, `ptmr8r.pl`, `ptmro8r.mtx` and `ptmro8r.pl`.

You will have to inform your device driver about the transformed font, using the syntax appropriate for that driver. For example, in `dvips` you add a line to `psfonts.map`:

```
ptmro8r Times-Roman ".167 SlantFont TeXBase1Encoding ReEncodeFont" <8r.enc
```

2.3 Miscellaneous settings

`\substitutesilent{ $\langle to \rangle$ }{ $\langle from \rangle$ }`
`\substitutenoisy{ $\langle to \rangle$ }{ $\langle from \rangle$ }`

This declares a \LaTeX font substitution, that the series or shape $\langle to \rangle$ should be substituted if necessary by the series or shape $\langle from \rangle$. `\substitutesilent` means that when the font substitution is made, no warning will given.

`\substitutenoisy` is the same as `\substitutesilent`, but gives a warning when the substitution is made by \LaTeX .

For example, to say that the series `bx` can be replaced by the series `b`, you say:

```
\substitutesilent{bx}{b}
```

To say that the shape `ui` can be replaced by the shape `i`, you say:

```
\substitutenoisy{ui}{it}
```

The following weight substitutions are standard:

```

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

```

The following shape substitutions are standard:

```

\substitutenoisy{ui}{it}
\substitutesilent{it}{sl}
\substitutesilent{sl}{it}

```

The `\installfontas` command should be considered as an alternative to using font substitution, as it gives much finer control over what `fd` entries will be made.

`\declaresize{<size>}{<fd-size-range>}`

This declares a new size, and gives the `fd` commands for it. For example, `fontinst.sty` declares the following sizes:

```

\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>}

```

`\declareencoding{<string>}{<etx>}`

This declares which `etx` file corresponds to which encoding string. For example, `fontinst.sty` declares the following encoding strings:

```

\declareencoding{TEX TEXT}{ot1}
\declareencoding{TEX TEXT WITHOUT F-LIGATURES}{ot1}
\declareencoding{TEX TYPEWRITER TEXT}{ot1tt}
\declareencoding{TEX MATH ITALIC}{oml}
\declareencoding{TEX MATH SYMBOLS}{oms}
\declareencoding{TEX MATH EXTENSION}{omx}
\declareencoding{EXTENDED TEX FONT ENCODING - LATIN}{t1}
\declareencoding{TEX TEXT COMPANION SYMBOLS 1---TS1}{ts1}
\declareencoding{TEXBASE1ENCODING}{8r}
\declareencoding{TEX TYPEWRITER AND WINDOWS ANSI}{8y}

```

2.4 Other

The following commands belong to this section, but there is currently no description of them here.

```
\afmtomtx{<afmfile>}{<mtxfile>}
\endrecordtransforms
\endreglyphfonts
\etxtopl{<encoding list>}{<plfile>}
\etxtovgl{<encoding list>}{<vglfile>}
\fakenarrow{<width factor>}
\generalpltomtx{<plfile>}{<mtxfile>}{<plsuffix>}{<opt-enc>}
\killglyph{<glyph>}
\killglyphweighted{<glyph>}{<weight>}
\latinfamily{<family>}{<commands>}
\mtxtomtx{<source MTX>}{<destination MTX>}
\mtxtopl{<mtxfile>}{<plfile>}
\NOFILES
\offmtxcommand{<command>}
\onmtxcommand{<command>}
\recordtransforms{<filename>}
\reglyphfont{<destination font>}{<source font>}
\reglyphfonts
\renameglyph{<to>}{<from>}
\renameglyphweighted{<to>}{<from>}{<weight>}
```

3 Mapmaking commands

The following commands belong to this section, but there is currently no description of them here. Note that several of them have a different syntax and meaning than they do in fontmaking command files.

```
\adddriver{<driver name>}{<fragment file name>}
\AssumeAMSBYYY
\AssumeBaKoMa
\AssumeLWFN
\AssumeMetafont
```



```

\debugvalue{<name>}
\declarepsencoding{<etx>}{<postscript name>}{<action>}
\donedrivers
\download{<file>}
\enctoetx{<encfile>}{<etxfile>}
\etxtoenc{<etxfile>}{<encfile>}
\fromafm{<AFM name>}{<PS name>}
\frompl{<PL name>}
\frommtx{<MTX name>}
\fromvpl
\makemapentry{<TEX font name>}
\reencodefont{<etx>}
\reglyphfont
\specifypsfont{<PS font name>}{<actions>}
\storemapdata{<TEX font name>}{<source>}{<transforms>}
\transformfont{<x-scale>}{<slant-scale>}

```

4 General commands

This section describes commands and mechanisms that are the same in all file types. Commands that are particular for one type of file are described in subsequent sections.

4.1 Variables

Many (but not all) of the activities fontinst perform can be understood as either “setting variables” or “formatting and writing to file data stored in some variable”. The accessing of variables is an important aspect of how fontinst works.

Variables come in different types and variables of different types live in different namespaces; `\int{foo}`, `\str{foo}`, and `\dim{foo}` refer to three different variables which are all named `foo`. Variables are either set or not set. Unless the contrary is stated explicitly, variables default to not being set. It is an error to access the value of a variable that has not been set. Fontinst variable assignments are as a rule local, i.e., will be undone when the enclosing `TeX` group is ended. Most command file commands that cause files to be read will begin a group before reading the file(s) and end the group at some point after having read them.

Taking string variables as an example, there are three commands for changing a string variable:

<code>\setstr{⟨name⟩}{⟨string expression⟩}</code> <code>\resetstr{⟨name⟩}{⟨string expression⟩}</code> <code>\unsetstr{⟨name⟩}</code>
--

The `\resetstr` command unconditionally sets the string variable `⟨name⟩` to the full expansion of the `⟨string expression⟩`. The `\unsetstr` command unconditionally renders the string variable `⟨name⟩` unset. If the the string variable `⟨name⟩` is currently unset then the `\setstr` command will set it to the full expansion of the `⟨string expression⟩`, but if it already is set then `\setstr` does nothing.

This pattern with three commands, one `\set...` which only sets unset variables, one `\reset...` which sets variables regardless of whether they have been set or not, and one `\unset...` which unsets variables is recurring in fontinst. Variables are most commonly set using some `\set...` command; this has the effect that the first command to try to set a variable is the one which actually sets it.

4.2 General commands

The following *general commands* can be used anywhere:

<code>\needsfontinstversion{⟨version⟩}</code>

This issues a warning if the current version of the fontinst package is less than `⟨version⟩`.

<code>\setdim{⟨dim⟩}{⟨dimension⟩}</code> <code>\setint{⟨int⟩}{⟨integer expression⟩}</code> <code>\setstr{⟨str⟩}{⟨string⟩}</code>
--

If the dimension variable `⟨dim⟩` is currently undefined, it is defined to be the current value of `⟨dimension⟩`.

If the integer variable `⟨int⟩` is currently undefined, it is defined to be the current value of `⟨integer expression⟩`.

If the string variable `⟨str⟩` is currently undefined, it is defined to be the current value of `⟨string⟩`.

<code>\setcommand{⟨command⟩}{⟨parameter text⟩}{⟨replacement text⟩}</code>

If the command `⟨command⟩` is currently undefined, it is defined to be the `⟨definition⟩`. This uses the same syntax for parameters as the \TeX `\def` command.

No. parameters	<code>⟨parameter text⟩</code>
0	<code>(empty)</code>
1	<code>#1</code>
2	<code>#1#2</code>
3	<code>#1#2#3</code>
and so on.	

<pre>\resetdim{⟨dim⟩}{⟨dimension⟩} \resetint{⟨int⟩}{⟨integer expression⟩} \resetstr{⟨str⟩}{⟨string⟩}</pre>
--

The dimension variable $\langle dim \rangle$ is defined to be the current value of $\langle dimension \rangle$.

The integer variable $\langle int \rangle$ is defined to be the current value of $\langle integer expression \rangle$.

The string variable $\langle str \rangle$ is defined to be the current value of $\langle string \rangle$.

<pre>\resetcommand{⟨command⟩}{⟨parameter text⟩}{⟨replacement text⟩}</pre>

The command $\langle command \rangle$ is defined to be the $\langle definition \rangle$, regardless of whether it was already defined or not. This is a synonym for the \TeX `\def` command.

<pre>\ifisint{⟨int⟩}\then \ifisdim{⟨dim⟩}\then \ifisstr{⟨str⟩}\then \ifisglyph{⟨glyph⟩}\then \ifiscommand{⟨command⟩}\then</pre>

Expands out to `\iftrue` if the integer variable $\langle int \rangle$ is defined, and `\iffalse` otherwise.

Expands out to `\iftrue` if the dimension variable $\langle dim \rangle$ is defined, and `\iffalse` otherwise.

Expands out to `\iftrue` if the string variable $\langle str \rangle$ is defined, and `\iffalse` otherwise.

Expands out to `\iftrue` if the glyph variable $\langle glyph \rangle$ is defined, and `\iffalse` otherwise.

Expands out to `\iftrue` if the command $\langle command \rangle$ is defined, and `\iffalse` otherwise.

<pre>\unsetdim{⟨dim⟩} \unsetint{⟨int⟩} \unsetstr{⟨str⟩} \unsetcommand{⟨command⟩}</pre>
--

Makes $\langle dim \rangle$, $\langle int \rangle$, $\langle str \rangle$, or $\langle command \rangle$ an undefined dimension, integer, string or command.

4.3 Integer expressions

The *integer expressions* provide a user-friendly syntax for \TeX arithmetic. They are used to manipulate any integers, including glyph dimensions, which are given in *afm* units, that is 1000 to the design size. \TeX p1 fonts have their dimensions converted to *afm* units automatically.

The *integer expressions* are:

`\number`

Returns the value of a TeX *number* (as explained in *The TeXbook*).

`\int{int}`

Returns the value of the integer variable *int*.

`\width{glyph}`
`\height{glyph}`
`\depth{glyph}`
`\italic{glyph}`

Returns the width, height, depth, or italic correction of the glyph variable *glyph*.

`\kerning{left}{right}`

Returns the kerning between the *left* and *right* glyph variables.

`\neg{integer expression}`
`\add{integer expression}{integer expression}`
`\sub{integer expression}{integer expression}`
`\max{integer expression}{integer expression}`
`\min{integer expression}{integer expression}`
`\mul{integer expression}{integer expression}`
`\div{integer expression}{integer expression}`
`\scale{integer expression}{integer expression}`
`\half{integer expression}`
`\otherhalf{integer expression}`

`\neg` returns the negation of the *integer expression*.

`\ad` returns the sum of the two *integer expression*s.

`\sub` returns the first *integer expression* minus the second.

`\mul` returns the product of the two *integer expression*s.

`\div` returns the first *integer expression* divided by the second.

`\scale` returns the first *integer expression* times the second, divided by 1000.

4.4 Other

The following commands belong to this section, but there is currently no description of them here.

`\begincomment`

`\bye`

`\Else`

```

\endcomment
\endfor(<name>)
\eTeX{<version number>}
\Fi
\fontinstcc
\fontinsterror{<subsystem>}{<error>}{<help>}
\fontinstinfo{<subsystem>}{<info>}
\fontinstwarning{<subsystem>}{<warning>}
\fontinstwarningnoline{<subsystem>}{<warning>}
\for(<name>){<start>}{<stop>}{<step>}
\foreach(<name>){<csep-list>}
\ifareglyphs{<glyph list>}\then
\ifiskern{<glyph1>}{<glyph2>}\then
\ifnumber{<integer expression>}{<rel>}{<integer expression>}\then
\ifoption{<string>}\then
\input <file name>
\messagebreak
\needsTeXextension{<extension tests>}{<who>}
\normalcc
\offcommand{<command>}
\oncommand{<command>}
\pdfTeX{<version number>}{<revision>}
\strint{<int>}

```

5 Encoding files

An *encoding file* (or `.etx` file) is a T_EX document consisting of:

```

\relax
ignored material
\encoding
<encoding commands>
\endencoding
ignored material

```

This describes the encoding of a font, using the *<encoding commands>*.

Since the encoding file ignores any material between `\relax` and `\encoding`, an *encoding file* can also be a L^AT_EX document.

See also the descriptions in `encspecs.tex`.

5.1 Encoding commands

The *encoding commands* are:

`\nextslot{integer expression}`

Sets the number of the next slot. If there is no `\nextslot` command, the number is the successor of the previous slot.

`\skipslots{integer expressions}`

Advances the number of the next slot.

New feature
v1.8

`\setslot{glyph}`
`slot commands`
`\endsetslot`

Sets the slot of the *glyph*. The *slot commands* describe the glyph, and give its usage in T_EX.

`\inputetx{file}`

Inputs the *encoding commands* of *file*.etx.

5.2 Slot commands

The *slot commands* are:

`\comment{text}`

A comment, which is ignored by fontinst.

`\label{text}`

A reference label. Ignored by fontinst.

`\ligature{ligtype}{glyph}{glyph}`
`\Ligature{ligtype}{glyph}{glyph}`
`\oddligature{note}{ligtype}{glyph}{glyph}`

Specifies a ligature of type *ligtype* from the current glyph followed by the first glyph to the second glyph. The *ligtype*s are as in vpl files (see the vptovf Web source for more details). For example:

```
\setslot{ff}  
  \ligature{LIG}{i}{ffi}  
  \ligature{LIG}{l}{ffl}
```

```
\comment{The ‘ff’ ligature.}
\endsetslot
```

```
\makerightboundary{<glyph>}
```

```
\Unicode{<code point>}{<name>}
```

```
\usedas{<type>}{<control sequence>}
```

Sets the T_EX control sequence for this slot, with the *type* taken from:

New feature
Obsolete?!

char	accent	mathord
mathbin	mathrel	mathopen
mathclose	mathpunct	mathvariable
mathaccent	mathdelim	

```
\nextlarger{<glyph>}
```

Sets a NEXTLARGER entry from the current slot to the *<glyph>*.

```
\vchar
<vchar commands>
\endvchar
```

Sets a VARCHAR entry for the current slot, using the *<vchar commands>*. The *<vchar commands>* are:

```
\vartop{<glyph>}
\varmid{<glyph>}
\varbot{<glyph>}
\varrep{<glyph>}
```

Sets the top, middle, bottom, or repeated *<glyph>* of the VARCHAR.

5.3 Other

The following commands belong to this section, but there is currently no description of them here.

```
\endsetleftboundary
\ifdirect
\ifisinslot{<glyph>}{<slot>}\then
\resetslotcomment{<text>}
\setfontdimen{<fontdimen no.>}{<integer variable>}
```

```

\setleftboundary{<glyph>}
\setslotcomment{<text>}
\useexamplefont{<font>}
\unsetslotcomment

```

6 Metric files

A *metric file* (or *.mtx* file) is a T_EX document consisting of:

<pre> \relax <i>ignored material</i> \metrics <metric commands> \endmetrics <i>ignored material</i> </pre>
--

This describes the glyphs in a font, using the *<metric commands>*.

Metric files are usually either *hand-crafted* or *transformable*. The transformable metric files typically encode the metrics of one particular font and are automatically generated. Hand-crafted metric files (such as `latin.mtx`) typically do not contain much explicit metric data, instead the code there makes use of metrics previously specified by other files to construct new glyphs or adjust metrics to meet special conditions. Whereas transformable metric files tend to be mere lists of metric data, the hand-crafted metric files are more like programs.

6.1 Metric commands

The *<metric commands>* are:

<pre> \setglyph{<name>} <glyph commands> \endsetglyph </pre>
--

If the glyph called *<name>* is undefined, it is built using the *<glyph commands>* given below, for example:

```

\setglyph{IJ}
  \glyph{I}{1000}
  \glyph{J}{1000}
\endsetglyph
\setglyph{Asmall}
  \glyph{A}{850}
\endsetglyph

```



```
\resetglyph{<name>}
<glyph commands>
\endsetglyph
```

Gives the definition of the glyph called *<name>* using the *<glyph commands>*.

```
\unsetglyph{<name>}
```

Makes the glyph called *<name>* undefined.

```
\setrawglyph{<name>}{<font>}{<dimen>}{<integer>}
{<integer>}{<integer>}{<integer>}{<integer>}
\setscaledrawglyph{<name>}{<font>}{<dimen>}{<integer>}{<integer>}
{<integer>}{<integer>}{<integer>}{<integer>}
```

This sets a glyph called *<name>* from the **, which has the given design size, slot, width, height, depth and italic correction. If the integer variable `rawscale` is set, the glyph will be scaled by that amount. This command will usually be generated automatically from an `afm` or `p1` file.

```
\setnotglyph{<name>}{<font>}{<dimen>}
{<integer>}{<integer>}{<integer>}{<integer>}
\setscalednotglyph{<name>}{<font>}{<dimen>}{<integer>}{<integer>}
{<integer>}{<integer>}{<integer>}{<integer>}
```

This sets a glyph called *<name>-not*, which is present in the **, but is not in the default encoding. It takes the same arguments as `\setrawglyph`, although the slot will normally be `-1`. This command will usually be generated automatically from an `afm` file.

```
\setkern{<glyph>}{<glyph>}{<integer expression>}
\resetkern{<glyph>}{<glyph>}{<integer expression>}
```

`\setkern` sets a kern between the two glyphs, scaled by the current value of `rawscale`, unless such a kern already has been set.

```
\setleftkerning{<glyph>}{<glyph>}{<integer expression>}
\setrightkerning{<glyph>}{<glyph>}{<integer expression>}
```

Sets the amount by which the first glyph should mimic how the second glyph kerns on the left or right, for example:

```
\setleftkerning{Asmall}{A}{850}
\setrightkerning{Asmall}{A}{850}
\setleftkerning{IJ}{I}{1000}
\setrightkerning{IJ}{J}{1000}
```

Sets the amount by which the first glyph should mimic how the second glyph kerns on the right, for example:

```
\setlefttrighthkerning{<glyph>}{<glyph>}{<integer expression>}
```

Sets the amount by which the first glyph should mimic how the second glyph
kerns on both sides, for example: New feature
v1.8

```
\setlefttrighthkerning{Asmall}{A}{850}
```

```
\noleftkerning{<glyph>}
\norightkerning{<glyph>}
\noleftrightkerning{<glyph>}
```

Removes all kerning on the specified side(s) of the *<glyph>*. New feature
v1.9

```
\inputmtx{<file>}
```

Inputs the *<metric commands>* of *<file>.mtx*.

6.2 Glyph commands

The *<glyph commands>* are:

```
\glyph{<glyph>}{<integer expression>}
```

Sets the named glyph *<glyph>* at the given scale, with 1000 as the natural size.
This:

- Advances the current glyph width.
- Sets the current glyph height to be at least the height of the named glyph,
adjusted for the current vertical offset.
- Sets the current glyph depth to be at least the depth of the named glyph,
adjusted for the current vertical offset.
- Sets the current glyph italic correction to be the same as the set glyph.

The named glyph must have already been defined, otherwise an error will occur. For example:

```
\setglyph{fi}
\glyph{f}{1000}
\glyph{i}{1000}
\endsetglyph
```

```
\glyphrule{<integer expression>}{<integer expression>}
```

Sets a rule of the given width and height, for example:

```
\setglyph{underline}
\glyphrule{333}{40}
\endsetglyph
```

`\glyphspecial{<text>}`

Sets a driver-dependent `\special`, for example:

```
\setglyph{crest}
  \glyphspecial{Filename: crest.eps}
\endsetglyph
```

`\glyphwarning{<text>}`

Sets a warning `\special`, and produces a warning message each time the glyph is used, for example:

```
\setglyph{missingglyph}
  \glyphrule{500}{500}
  \glyphwarning{Missing glyph 'missingglyph'}
\endsetglyph
```

`\moveright{<integer expression>}`

Moves right by the given amount, and advances the current glyph width, for example:

```
\setglyph{A small}
  \moveright{50}
  \glyph{A}{700}
  \moveright{50}
\endsetglyph
```

`\moveup{<integer expression>}`

Moves up by the given amount, and advances the current vertical offset. Each glyph should always end at vertical offset zero, for example:

```
\setglyph{onehalf}
  \moveup{500}
  \glyph{one}{700}
  \moveup{-500}
  \glyph{slash}{1000}
  \moveup{-200}
  \glyph{two}{700}
  \moveup{200}
\endsetglyph
```

`\push`
`<glyph commands>`
`\pop`

Performs the `<glyph commands>` without adjusting the current position or glyph width, for example:

```

\setglyph{aacute}
\push
\moveright{\div{\sub{\width{a}}{\width{acute}}}{2}}
\glyph{acute}{1000}
\pop
\glyph{a}{1000}
\endsetglyph

```

`\glyphpcc{<glyph>}{<integer expression>}{<integer expression>}`

This is generated from PCC instructions in an afm file, and is syntactic sugar for:

```

\push
\moveright{<first integer expression>}
\moveup{<second integer expression>}
\glyph{<glyph>}{1000}
\pop

```

`\resetwidth{<integer expression>}`
`\resetheight{<integer expression>}`
`\resetdepth{<integer expression>}`
`\resetitalic{<integer expression>}`

Sets the width, height, depth, or italic correction of the current glyph.

`\samesize{<glyph>}`

Sets the dimensions of the current glyph to be the same as *<glyph>*.

Inside the definition of *<glyph>*, you can use expressions such as `\width{<glyph>}`, which will refer to the glyph defined so far. For example, a display summation sign can be defined to be a text summation \sum scaled 120% with 0.5 pt extra height and depth using:

```

\setglyph{summationdisplay}
\glyph{summationtext}{1200}
\resetheight{\add{\height{summationdisplay}}{50}}
\resetdepth{\add{\depth{summationdisplay}}{50}}
\endsetglyph

```

Within a `\resetglyph`, these expressions will refer to the previous definition of the glyph. For example, you can add sidebearings to the letter ‘A’ with:

```

\resetglyph{A}
\moveright{25}
\glyph{A}{1000}
\moveright{25}
\endresetglyph

```

6.3 Other

The following commands belong to this section, but there is currently no description of them here.

```
\aliased{<font's name>}{<alias name>}  
\ProvidesMtxPackage{<package name>}  
\unsetkerns{<left glyph list>}{<right glyph list>}  
\usemtxpackage{<package list>}
```

7 fontdoc commands

```
\macroparameter{<digit>}  
\plaindiv  
\plainint  
\plainmax  
\plainmin  
\showbranches  
\slotexample  
\textunicode{<code point>}{<name>}
```

8 fontinst variables

The following is a list of the fontinst variables that are accessible for the user through the `\set...`, `\reset...`, `\unset...`, etc. commands. You may of course set or use other variables in the MTX and ETX files you write yourself, as does for example the standard MTX file `latin.mtx`, but all variables that fontinst commands implicitly use or set are listed below.

accapheight (integer denoting length)
 Description The height of accented full capitals.
 Set by MTX files.
 Used by Some ETX and MTX files.

address (string)
 Description Snailmail address put in BibTeX-style file header of automatically generated ENC files. No **address** field is written unless the **address** string is set. Quotes are not automatically inserted around the address string.
 Set by ETX files.
 Used by The ETX-to-ENC converter.

afm-name (string)
 Description Name of source font. Internal variable.

Set by `\from...` commands.

Used by The `\transformfont`, `\installfont`, `\installrawfont`, and `\reglyphfont` commands.

ascender (integer denoting length)

Description The ascender height of the font.

Set by MTX files. The AFM-to-MTX converter usually writes `\setint` commands for this integer.

Used by Some MTX and ETX files.

author (string)

Description Author name(s) put in Bib \TeX -style file header of automatically generated ENC files. See the macro `\ref_to_sourcefile` for more details.

Set by ETX files.

Used by The ETX-to-ENC converter. When not set, the value "See file *<etx name>*" is used instead.

\autoinstallfamily (command)

Description Command called by the font installation commands, as

$$\text{\autoinstallfamily}\{\langle encoding \rangle\}\{\langle family \rangle\}$$

when they are asked to install a font with a combination of *<encoding>* and *<family>* that has not been seen before (there was no explicit `\installfamily`).

Set by Explicit commands. Defaults to calling `\installfamily`.

Used by Font installation commands.

axisheight (integer denoting length)

Description Math formula parameter σ_{22} .

Set by MTX files.

Used by Some ETX and MTX files.

baselineskip (integer denoting length)

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

Set by MTX files.

Used by Some ETX and MTX files.

bigopspacing1 (integer denoting length)

Description Math formula parameter ξ_9 .

Set by MTX files.

Used by Some ETX and MTX files.

bigopspacing2 (integer denoting length)

Description Math formula parameter ξ_{10} .

Set by MTX files.

Used by Some ETX and MTX files.

bigopspacing3 (integer denoting length)

Description Math formula parameter ξ_{11} .

Set by MTX files.

Used by Some ETX and MTX files.

bigopspacing4 (integer denoting length)

Description Math formula parameter ξ_{12} .

Set by MTX files.

Used by Some ETX and MTX files.

bigopspacing5 (integer denoting length)

Description Math formula parameter ξ_{13} .
Set by MTX files.
Used by Some ETX and MTX files.

`capheight` (integer denoting length)
Description The height of the font's full capitals.
Set by MTX files. The AFM-to-MTX converter usually writes `\setint` commands for this variable.
Used by Some MTX and ETX files.

`codingscheme` (string)
Description The codingscheme name.
Set by ETX files.
Used by The (V)PL writer. When not set, the value UNKNOWN is used instead.

`defaultrulethickness` (integer denoting length)
Description Math formula parameter ξ_8 .
Set by MTX files.
Used by Some ETX and MTX files.

`delim1` (integer denoting length)
Description Math formula parameter σ_{20} .
Set by MTX files.
Used by Some ETX and MTX files.

`delim2` (integer denoting length)
Description Math formula parameter σ_{21} .
Set by MTX files.
Used by Some ETX and MTX files.

`denom1` (integer denoting length)
Description Math formula parameter σ_{11} .
Set by MTX files.
Used by Some ETX and MTX files.

`denom2` (integer denoting length)
Description Math formula parameter σ_{12} .
Set by MTX files.
Used by Some ETX and MTX files.

`descender` (integer denoting length)
Description The depth of lower case letters with descenders.
Set by MTX files.
Used by Some ETX and MTX files.

`descender_neg` (integer denoting length)
Description The vertical position of the descender line of the font, i.e., the negative of the font's descender depth.
Set by MTX files. The AFM-to-MTX converter usually writes `\setint` commands for this variable.
Used by Some MTX and ETX files.

`designsize` (dimension)
Description The design size of the font.
Set by MTX files. The (V)PL-to-MTX converter usually writes `\setdim` commands for this variable.
Used by The (V)PL writer. The design size defaults to 10 pt if this variable is not set.

Note The value of this variable has no effect on how the font is declared to L^AT_EX.

designunits (dimension denoting a real number)
Description The design size of a font expressed in the design unit used in a (V)PL file.
Set by MTX files. The (V)PL-to-MTX converter usually writes `\setdim` commands for this variable.
Used by Nothing. If this variable is set, but to any value other than 1 pt, then some metrics are most likely wrong.

digitwidth (integer denoting length)
Description The median width of the digits in the font.
Set by MTX files.
Used by Some ETX and MTX files.

email (string)
Description Email address put in BibT_EX-style file header of automatically generated ENC files. See the macro `\ref_to_sourcefile` for more details.
Set by ETX files.
Used by The ETX-to-ENC converter. When not set, the value "See file *<etx name>*" is used instead.

encodingname (string)
Description The name by which the encoding in question is made known to a Postscript interpreter.
Set by ETX files.
Used by The ETX-to-ENC converter. When not set, the value `fontinst-autoenc-<etx name>` is used instead.

etx-name (string)
Description Name of ETX file. Internal variable in `\transformfont`.
Set by The `\reencodefont` command.
Used by The `\mtxtomtx` command.

extraspace (integer denoting length)
Description The natural width of extra interword glue at the end of a sentence.
Set by MTX files.
Used by Some ETX and MTX files.

fontdimen(*n*) (integer)
Description Family of semi-internal variables that store the values to use for font dimension *n*. It is preferred that the newer `\setfontdimen` interface is used for setting these values.
Set by ETX files.
Used by The (V)PL writer.

\iftokeep (macro)
Description `\iftokeep #1 \then`, where *#1* will be a *<number>*, behaves like a switch and decides whether a glyph is kept or not while re-glyphing.
Set by Explicit commands. Defaults to

$$\text{\iftokeep \#1 \then} \mapsto \text{\ifnum -1<\#1}$$

Used by The `\reglyphfont` command.

interword (integer denoting length)
Description The natural width of interword glue (spaces).
Set by MTX files.
Used by Some ETX and MTX files.

italicslant (integer denoting factor)
Description The italic slant of a font.
Set by MTX files generated from AFM or (V)PL files. MTX files generated by `\transformfont`. Locally in the AFM-to-MTX converter for possible use in `\uprightitalcorr` or `\slanteditalcorr`.
Used by MTX files (`latin.mtx` and the like). ETX files (for determining `fontdimen(1)`).

killweight (integer)
Description Weight for glyphs that are killed.
Set by Explicit commands. Defaults to `-10` if not set.
Used by The `\killglyph` command; indirectly the `\reglyphfont` command.

letterspacing (integer denoting length)
Description Extra width added to all glyphs of a font.
Set by ETX or MTX files.
Used by The (V)PL writer. Defaults to `0` if not set.

maxdepth (integer denoting length)
Description The maximal depth over all slots in the font.
Set by MTX files.
Used by Some ETX and MTX files.

maxdepth_neg (integer denoting length)
Description The negative of the maximal depth of a glyph in the font.
Set by MTX files. The AFM-to-MTX converter usually writes `\setint` commands for this variable.
Used by Some ETX and MTX files.

maxheight (integer denoting length)
Description The maximal height of a glyph in the font.
Set by MTX files. The AFM-to-MTX converter usually writes `\setint` commands for this variable.
Used by Some ETX and MTX files.

minimumkern (integer denoting length)
Description Kerns whose size in absolute value is less than or equal to this variable are ignored.
Set by Command files or MTX files.
Used by The AFM-to-MTX converter and the (V)PL file generator. When not set, the value `0` is used instead.

monowidth (flag integer)
Description Set if this font is monowidth, unset otherwise.
Set by MTX files. The AFM-to-MTX converter writes a `\setint` command for this variable if the AFM specifies `IsFixedPitch true`.
Used by Some MTX files (`latin.mtx` and the like), ETX files.

num1 (integer denoting length)
Description Math formula parameter σ_8 .
Set by MTX files.
Used by Some ETX and MTX files.

num2 (integer denoting length)

Description Math formula parameter σ_9 .
Set by MTX files.
Used by Some ETX and MTX files.

num3 (integer denoting length)
Description Math formula parameter σ_{10} .
Set by MTX files.
Used by Some ETX and MTX files.

quad (integer denoting length)
Description The quad width of the font, normally approximately equal to the font size and/or the width of an ‘M’.
Set by MTX files.
Used by Some ETX and MTX files.

PSfontsuffix (string)
Description Suffix added to font names to form name of file to download to include font.
Set by Explicit commands in mapmaking command files. Defaults to ‘.pfa’.
Used by The map file fragments writer.

rawscale (integer denoting factor)
Description Scaling factor applied to raw glyphs.
Set by The `\installfont` command (scaled clauses in argument #2). Unset for metric files listed without a scaled clause.
Used by The `\setrawglyph`, `\setnotglyph`, `\setscaledrawglyph`, `\setscalednotglyph`, `\setkern`, and `\resetkern` commands.

renameweight (integer)
Description Weight for glyphs that are renamed.
Set by Explicit commands. Defaults to 1 if not set.
Used by The `\renameglyph` command; indirectly the `\reglyphfont` command.

requireglyphs (flag integer)
Description Set if warnings are to be generated for glyphs listed in ETX files but not present in the glyph base.
Set by Explicit commands. By default not set.
Used by The (V)PL file generator.

rightboundary (string)
Description The name of a glyph with the property that kerns on the left may be intended as right word boundary kerns.
Set by MTX files. The (V)PL-to-MTX converter can write `\setstr` commands for this variable.
Used by Some MTX files.

shrinkword (integer denoting length)
Description The (finite) shrink component of interword glue.
Set by MTX files.
Used by Some ETX and MTX files.

slant-scale (integer denoting factor)
Description Factor to slant by. Internal variable in `\transformfont`.
Set by The `\slantfont`, `\xscalefont`, and `\scalefont` commands.
Used by The `\mtxtomtx` command.

\SlantAmount (macro expanding to an integer expression)
Description Slant factor used for faking oblique shape.

Set by Explicit commands. Defaults to 167.

Used by The `\latinfamily` command.

`\slanteditalcorr` (macro expanding to an integer expression)

Description The integer expression used to calculate a guess for the italic correction of glyphs in a font with positive slant. It has the syntax

`\slanteditalcorr {<width>}{<left>}{<right>}{<bottom>}{<top>}`

where *<width>* is the glyph's advance width, and the remaining arguments are coordinates of sides of the glyph's bounding box. The `italicslant` integer provides the italic slant of the font.

Set by Explicit commands in fontinst command files. Defaults to

$$\max\{0, right - width\}.$$

Used by The AFM-to-MTX converter.

`stretchword` (integer denoting length)

Description The (finite) stretch component of interword glue.

Set by MTX files.

Used by Some ETX and MTX files.

`sub1` (integer denoting length)

Description Math formula parameter σ_{16} .

Set by MTX files.

Used by Some ETX and MTX files.

`sub2` (integer denoting length)

Description Math formula parameter σ_{17} .

Set by MTX files.

Used by Some ETX and MTX files.

`subdrop` (integer denoting length)

Description Math formula parameter σ_{19} .

Set by MTX files.

Used by Some ETX and MTX files.

`sup1` (integer denoting length)

Description Math formula parameter σ_{13} .

Set by MTX files.

Used by Some ETX and MTX files.

`sup2` (integer denoting length)

Description Math formula parameter σ_{14} .

Set by MTX files.

Used by Some ETX and MTX files.

`sup3` (integer denoting length)

Description Math formula parameter σ_{15} .

Set by MTX files.

Used by Some ETX and MTX files.

`supdrop` (integer denoting length)

Description Math formula parameter σ_{18} .

Set by MTX files.

Used by Some ETX and MTX files.

`TFMfileprefix` (string)

Description Prefix (typically a path) added to names of TFM files.

Set by Explicit commands in mapmaking command files. By default not set, which is equivalent to being empty.

Used by The PLtoTF “map file fragments writer”.

underlinethickness (integer denoting length)
Description The recommended thickness of an underlining rule.
Set by MTX files. The AFM-to-MTX converter usually writes `\setint` commands for this variable.
Used by Some MTX files (`latin.mtx` and the like).

\uprightitalcorr (macro expanding to an integer expression)
Description The integer expression used to calculate a guess for the italic correction of glyphs in a font with non-positive slant. It has the syntax
`\uprightitalcorr {<width>}{<left>}{<right>}{<bottom>}{<top>}`
 where `<width>` is the glyph’s advance width, and the remaining arguments are coordinates of sides of the glyph’s bounding box. The `italicslant` integer provides the italic slant of the font.
Set by Explicit commands in `fontinst` command files. Defaults to 0.
Used by The AFM-to-MTX converter.

version (string)
Description Version number put in BibTeX-style file header of automatically generated ENC files. See the macro `\ref_to_sourcefile` for more details.
Set by ETX files.
Used by The ETX-to-ENC converter. When not set, the value “See file `<etx name>`” is used instead.

verticalstem (integer denoting length)
Description The dominant width of vertical stems (usually the width of stems of lower case letters).
Set by MTX files. The AFM-to-MTX converter writes `\setint` commands for this variable if the AFM file specifies `StdVW`.
Used by Currently nothing.

warningspecials (switch)
Description Controls whether `\glyphwarning` commands will generate VPL SPECIALs. Defaults to ‘true’.
Set by Explicit commands (`\warningspecialstrue` and `\warningspecialsfalse`).
Used by The (V)PL file generator.

x-scale (integer denoting factor)
Description Horizontal scaling factor. Internal variable in `\transformfont`.
Set by The `\xscalefont` and `\scalefont` commands.
Used by The `\mtxtomtx` command.

xheight (integer denoting length)
Description The x-height of the font.
Set by MTX files. The AFM-to-MTX and (V)PL-to-MTX converters usually write `\setint` commands for this variable.
Used by MTX files, and ETX files (for determining `fontdimen(5)`).

y-scale (integer denoting factor)
Description Vertical scaling factor. Internal variable in `\transformfont`.
Set by The `\yscalefont` and `\scalefont` commands.
Used by The `\mtxtomtx` command.

<glyph>-spacing (integer denoting length)

Description Glyph-specific override for `letterspacing`; extra width added to the glyph *⟨glyph⟩* as part of the process of writing a VPL file.

Set by ETX or MTX files.

Used by The (V)PL writer. Defaults to 0 if not set.

Besides these, the `\latinfamily` command provides a whole range of parameters (`\latin_weights`, `\latin_widths`, `\latin_shapes`, etc.) that are often used somewhat like variables. That subject does however deserve to be treated separately.

9 Customisation

The `fontinst` package reads a file `fontinst.rc` if it exists. This can contain your own customisations.

You can create a `fontinst` format by running `iniTeX` on `fontinst.sty` then saying `\dump`.

10 Notes on features new with v 1.9

The following notes are copied from `fisource.tex`; they were written to explain new `fontinst` features to old `fontinst` users.

10.1 Metric packages

`Fontinst` has traditionally come with a collection of MTX files that complement the MTX files generated from base font metrics, in that they build glyphs that may be missing from the base fonts or in some other way needs to be improved. The most well-known of these is the `latin.mtx` file; other examples include `textcomp.mtx`, `mathit.mtx`, and `latinsc.mtx`. A problem with these is however that they cannot produce optimal results for all fonts simply because there are irregular differences in how fonts are set up by the foundries. Most glyphs come out all right, but there are usually a few for which the parameters used are more or less wrong. Therefore most high quality font installations are made with modified versions of these files, where the parameters have been tuned to the specific font design.

Modifying in particular `latin.mtx` is however not an entirely easy task, because this is a rather large file (with plenty of archaic pieces of code in curious places). Doing it once is no big problem, but if one has to do it several times (maybe because some errors are discovered in the original `latin.mtx`) then it is probably no fun anymore. Furthermore, if one has two or three modified copies of this file because one has made high quality installations of that many different fonts then even a trivial bugfix might start to feel like entirely too much work.

If one has to make modifications then it is usually easier to deal with several small files (many of which can be used unchanged) than one big file. Thus

it would be better if these big files were split up into several smaller ones. The main problem with splitting up something like `latin.mtx` 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 defined 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 mentions 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. The `newlatin.mtx` file is an alternative to `latin.mtx` that implements this strategy. Most of the actual code is located in the following metric packages:

<code>ltxcmds.mtx</code>	Defines some common commands used by the other files.
<code>llbuild.mtx</code>	Builds the latin lower case alphabet (unaccented letters are ‘unfakable’, the rest are constructed if not present in the base fonts).
<code>lubuild.mtx</code>	Builds the latin upper case alphabet.
<code>lsbuild.mtx</code>	Builds accented letters in the latin smallcaps alphabet, but only if there are unaccented letters to build them from in the base fonts.
<code>lsfake.mtx</code>	Fakes a latin smallcaps alphabet by shrinking the upper case alphabet, but only if the glyph had not already been manufactured.
<code>lsmisc.mtx</code>	Make some miscellaneous smallcaps glyphs (mostly “smallcaps f-ligatures”).
<code>ltpunct.mtx</code>	Makes digits, punctuation marks, and other symbols (mostly by marking as “unfakable”).

All of these are easy to use as components of equivalents of a modified `latin.mtx` files, and all dependencies of one package upon another are handled via explicit `\usemtxpackage` commands.

10.2 Word boundary ligatures and kerns

One of the new features added in T_EX 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, then `\ifisglyph{g-not}` 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.

10.3 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 glyphs 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” (SC) 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
\setlefttrighthighkerning{Asmall}{a}{1000}
\unsetglyph{a}
\nolefttrighthighkerning{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 occurrences 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 mentions. The details of this mechanism are however somewhat tricky, so those interested in the full generality should read the comments in the source of `fontinst`. 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`.

A special rule pertains to the `\setrawglyph`, `\setnotglyph`, `\setscaledrawglyph`, and `\setscalednotglyph` commands, since `\transformfont` doesn't care what something was in the source font when it generates the transformed font. To turn these commands off while reglyphing, you use `\offmtxcommand` on `\setscaledrawglyph`.

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 SC 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 SC 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.

10.4 Making map file fragments

A *map file fragment* is the lines¹ of a map file that the corresponding driver would need for handling some set of fonts. When told to, `fontinst` can (in a fairly automatic way) create the map file fragment which is needed for the set of raw fonts `fontinst` has (i) installed directly (using `\installrawfont`) or (ii) used as a base font for some installed virtual font (generated by `\installfont`). `Fontinst` does not support the map file syntaxes of every existing driver, but the system is designed to be extendable and contributions that extend its capabilities are welcome. Nor can `fontinst` examine your \TeX system and determine every piece of information needed to make the correct map file fragments, but you can tell it roughly how your installation looks, it can make guesses which work most of the time, and you can specify most things explicitly if the guesses turn out to be wrong. Should the available options for configuring the process turn out to be inadequate for your needs, then please write to the `fontinst` mailing list about this—there is probably a way to improve the system so that your needs can be met.

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

```
\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):

<code>\input finstmsc.sty</code>	% Input command definitions
<code>\general settings</code>	% See below
<code>\adddriver{<driver name>}{<output file>}</code>	% Open output file
<code>\input whatever.tex</code>	% Writes to output file(s)
<code>\donedrivers</code>	% Close output file(s), tidy up
<code>\bye</code>	% Quit

¹Not in general an entire map file, hence the word *fragment*.

The `\adddriver` command gives the order “write map file entries for the *<driver name>* DVI driver to the file *<output file>*.” The plan is that it should be possible to use the name of just about any major driver (`dvips`, `xdvi`,² `pdftex`,³ `0zTeX`, etc.) here and get suitable map file entries for that driver as output, but for the moment only the `dvips` and `dvipdfm`⁴ drivers are supported.

You may also use `debug` or `pltotf` for *<driver name>*. The `debug` “DVI driver” file simply contains 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. It could be the right choice if you’re going to write the map file manually, as the combined effects of several font transformations are not always easy to compute manually. The file generated for the `pltotf` “driver” is actually a shell script consisting of a sequence of `pltotf` commands. These commands perform the `pl` to `tfm` conversion for precisely those fonts that are actually needed (`fontinst` usually generates `pl` files also for a number of fonts at intermediate stages of transformation, and many of these need not be converted to `tfm` files). The `TFMfileprefix` string can be used to add a directory path to the `tfm` file names, perhaps saving the step of moving them to their proper location later.

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. If some metric file you have used contains `\setrawglyph` commands that were not automatically generated by `fontinst`, then there might not be a `\storemapdata` for the font they refer to in `whatever.tex`, so you will have to include such a command yourself somewhere. This can for example be done in the *<general settings>* part of the above example file.

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 your \TeX system, about the set of fonts you are using on this run, or about something else that might be useful. Some such commands are of a general nature and affect what assumptions `fontinst` will make in certain conditions when no specific information is available. For the moment there commands are:

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

`\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.

²Or does that use the same map file as `dvips`? I heard somewhere that it did. /LH

³`pdfTeX` can read the map files generated for `dvips`, but a separate driver is desirable because the formats are not completely identical.

⁴Whose support I made very much to illustrate that you *don’t* have to be a big and ancient driver like `dvips` to have supporting code put into `fontinst`. (The fact that I just happened to have printed out the documentation and that it was easy to read also helped, of course.) Note, however, that there won’t be any support for a driver unless someone sits down and writes the code for it! Don’t assume I will. /LH

Otherwise the default action of the routine for finding out the postscript name of a font simply is to observe that it hasn't got a clue about what the right value is when the metrics were taken from a PL file, and therefore it writes '?????' for the postscript name.

`\AssumeLWFN` Assume postscript fonts for which nothing else has been specified are stored in files which are named according to the MacOS scheme for LWFNs.

Otherwise the default action is to use the name of the AFM or PL from which the metrics were originally taken, and add the file suffix stored in the string `PSfontsuffix`. The default value of this string is `.pfa`, but it can be changed using `\resetstr`.

If neither the default nor the LWFN scheme produce correct results then you may use the more specific `\specifypsfont` command, which describes exactly which file (or files, if any) a given font is stored in. The syntax of this command is

```
\specifypsfont{<PS font name>}{<actions>}
```

where the `<actions>` is a sequence of "action commands". Currently the only such command is

```
\download{<file>}
```

which instructs the map file writer to include in any entry using that PS font and "instruction" that the specified file should be downloaded. Some examples are

```
\specifypsfont{Times-Roman}{%
\specifypsfont{Shareware-Cyrillic-Regular}{\download{fcyr.gsf}}
\specifypsfont{zmn18ac6}{%
\download{MinionMM.pfb}\download{zmn18ac6.pro}%
}
```

Many dvi drivers (for example dvips) have more than one style of font downloading (e.g., partial and full downloading). This interface could be extended to control also such finer details (for example by adding a `\fulldownload` command to force full download of a font), but requests for this has so far been scarce.

Finally, there is the `\declarepsencoding` command which is used to link ETX files to postscript encodings. If no postscript encoding has been linked to a given ETX file then `fontinst` will automatically create a postscript encoding (`.enc`) file for that encoding, and use this file for all reencoding commands. The `8r` encoding is predeclared, and it doesn't matter if an encoding is undeclared if you never use it to reencode fonts, but there is potentially a problem with not having declared encodings you have installed and use for reencoding, as you may then find yourself having two files with identical names that define encodings that do not have the same name (as far as postscript is concerned).

10.5 Tuning accent positions—an application of loops

The accent placements made by `latin.mtx` certainly aren't perfect for all fonts, and the only way to find out where they should be put is through trying in text the accented letters you get for a couple of values for the position parameter and deciding which one works best. Since to try one parameter value you need to (i) edit it into an MTX file, (ii) run `fontinst`, (iii) run `vptovf`, (iv) run \TeX on some test text, and (v) print that text, trying one parameter value can take annoyingly much time. Repeating the same procedure ten times to test ten values is not something one does without being bored (unless one scripts it, of course), but it is possible to try ten parameter values in a single virtual font, and without doing very much typing.

Say you're not too happy with how `latin.mtx` positions the accent in the `ohungarumlaut` glyph:

```
\setglyph{ohungarumlaut}
\topaccent{o}{hungarumlaut}{500}
\endsetglyph
```

The 500 is the horizontal position (in thousandths of the width of the o) that the centre of `hungarumlaut` in the glyph constructed will have, so that is the position parameter value that you want to change. Create an MTX file containing the code

```
\for(pos){250}{750}{50}
\setglyph{ohungarumlaut\strint{pos}}
\topaccent{o}{hungarumlaut}{\int{pos}}
\endsetglyph
\setlefttrighthkerning{ohungarumlaut\strint{pos}}
{ohungarumlaut}{1000}
\endfor(pos)
```

This will set eleven glyphs `ohungarumlaut250`, `ohungarumlaut300`, `ohungarumlaut350`, ..., `ohungarumlaut750`, each being an Hungarianly umlauted 'o' (i.e., an 'ő') but all having that umlaut in slightly different positions. In order to put them in a font, you also need to make an encoding that contains them. Therefore create an ETX file which contains the code

```
\relax\encoding
\nextslot{"C0}
\for(pos){250}{750}{50}
\setslot{ohungarumlaut\strint{pos}}
\endsetslot
\endfor(pos)
\endencoding
```

The command for installing this experiment font would be something like

```
\installfont{<some name>}{<the normal list of metrics>,<the new MTX>}
{ot1,<the new ETX>}{OT1}...
```

The reason for including `ot1` in the third argument above is that you'll need letters other than 'ő' against which you can compare the experimental glyphs.

It would not have been possible to use `t1` instead of `ot1` (even though that has more Hungarian letters) since that would set all slots in the font and leave none for these experimental `ohungarumlauts`.

It is even possible to use a loop for making the test text. The \LaTeX macros

```
\newcount\slotcount
\newcommand\testtext[3]{%
  \slotcount=#1\relax
  \begin{description}%
    \loop\item[\the\slotcount]#3%
    \ifnum #2>\slotcount \advance \slotcount 1 \repeat
  \end{description}%
}
\DeclareTextCompositeCommand{\H}{OT1}{o}{\char\slotcount}
```

will let you write

```
\testtext{\langle first \rangle}{\langle last \rangle}{\langle text \rangle}
```

to get the text $\langle text \rangle$ typeset once for each slot from $\langle first \rangle$ to $\langle last \rangle$ inclusive, with \H{o} ranging through the glyphs in this interval. Thus in this case `\testtext{"CO}{CA}{Erd\H{o}s}` would be a trivial test.

10.6 Font installation commands

The `\installfont`, `\installrawfont`, and `\installfontas` commands have the respective syntaxes

```
\installfont{\font-name}{\metrics}{\etx-list}
  {\encoding}{\family}{\series}{\shape}{\size}
\installrawfont{\font-name}{\metrics}{\etx-list}
  {\encoding}{\family}{\series}{\shape}{\size}
\installfontas{\font-name}
  {\encoding}{\family}{\series}{\shape}{\size}
```

The $\langle font-name \rangle$ argument and the last five arguments are common to all these commands. The first argument is the name of a \TeX font to install. The last five arguments are the NFSS attributes under which that font will be declared to \LaTeX —encoding, family, series, shape, and size. It is worth observing that encoding names are usually in upper case, whereas the family, series, and shape are usually in lower case. The size argument is either a shorthand (declared using `\declaresize`) for a particular font size (or range of font sizes), or an explicit list of font sizes or ranges of sizes, which is copied directly to the font declaration. The most common case is to let the size argument be empty, as that is declared as a shorthand for “any size”.

The `\installfontas` command does not itself create the font, it just makes a note that the specified font declaration should be written to the proper FD file at `\endinstallfonts`. The `\installfont` and `\installrawfont` commands do however produce the font, in the sense that they write a VPL and PL respectively file for the font. It depends solely on the $\langle metrics \rangle$ and $\langle etx-list \rangle$ arguments what this font will contain. Many features of these arguments are new with `fontinst v 1.9`; therefore the complete syntaxes are described below.

Both arguments are comma-separated lists of basically file names (not including an extension). The files listed in the $\langle metrics \rangle$ are font metric files which together build up a *glyph base* (definitions of glyphs and metrics related to one or several glyphs), whereas the files listed in the $\langle etx-list \rangle$ are encoding definition files that select a subset of the glyph base for turning into a T_EX font. The font metrics can be in either of the four formats MTX, PL, AFM, and VPL, which are considered in that order. If the metrics are not originally in MTX format then they will be converted to this format (a new file will be created) before they are used. The encoding definitions must be in ETX format. The files actually read will have a suffix `.mtx`, `.pl`, `.afm`, `.vpl`, or `.etx` appended to the name given, depending on which format is expected.

Within each element of the comma-separated list, the actual file name is followed by zero or more *modifier clauses*. A *modifier clause* consists of a *keyword* followed by some number (usually one) of *arguments*, separated by spaces. The whole thing looks a lot like the *rule specifications* of e.g. the `\vrule` command, but here the spaces are mandatory. The currently defined *modifier clause*s are

`\option{string}` Available for metric and encoding files. This adds $\langle string \rangle$ to the list of options for this file, which may affect what code the file executes. The file can then test, using the `\ifoption` command, whether a specific string is one of the options it was given.

`\scaled{factor}` Available for metric files. Causes the `rawscale` integer variable to be set to the $\langle factor \rangle$ (an integer expression) while the file is being read. This scales glyphs and kerns that are added to the glyph base by the $\langle factor \rangle$.

`\suffix{suffix}` Available for metric files. Causes $\langle suffix \rangle$ to be appended to every glyph name appearing in a glyph or kern that file adds to the glyph base. Thus “`suffix /2`” effectively changes a

`\setrawglyph{a}...`

to a

`\setrawglyph{a/2}...`

`\encoding{etx-name}` Available for metric files, and forces `fontinst` to only consider the PL and VPL formats for this font. As these file formats do not contain glyph names, an ETX file is used to assign glyph names to the slots in the font. This ETX file is usually selected according to the `CODINGSCHEME` property of the PL or VPL (using the correspondences set up via the `\declareencoding` command), but that information is not always as one would want it (there are even fonts for which it is quite wrong). An encoding clause bypasses this automatic mechanism, so that the file $\langle etx-name \rangle$.`etx` is used instead.

`\mtxasetx` This is available for files in the $\langle etx-list \rangle$. The actual function of a

$\langle file-name \rangle$ `mtxasetx`

item in the $\langle etx-list \rangle$ is that the file $\langle file-name \rangle$.`mtx` is inputted (not $\langle file-name \rangle$.`etx`) and that the correspondence between glyph names and slot numbers

set up in `\setrawglyph` or `\setscaledrawglyph` commands in this file is treated as if it had been set up by `\setslot` commands in an ETX file. Provided the MTX file is transformable, the glyph base will be unaffected.

The purpose of this feature is to simplify quick and dirty installations of odd fonts for which no suitable ETX file is available. This can be useful in early stages of the design of a new font, but is inferior to installation using proper ETX files since one for example cannot specify any ligatures in MTX files.

Furthermore there is a special exception for the $\langle metrics \rangle$: if the first token in one of the list items is the control sequence `\metrics`, then the rest of that item is interpreted as explicit metric commands to execute.

If the $\langle metrics \rangle$ of two subsequent `\installfont` or `\installrawfont` commands are identical then the glyph bases will be identical as well. This creates an opportunity for optimization, which fontinst makes use of by caching glyph bases from one installation command to the next so that the glyph base does not have to be rebuilt in these cases. A side-effect of this caching is that local assignments made between two font installation commands are cleared out with the glyph base, but `\setint` and similar fontinst commands make global assignments when used in such positions.

Some examples might be in order. The first is an adaptation of an installation command from `mfnt-0.59` by Matthias Clasen and Ulrik Vieth: the installation command for the 8-bit math font `xma1000` (which can be thought of as being to `cmmi10` sort of as `ecrm1000` is to `cmr10`). The first three encoding clauses are more fine-tuning—without them, a few glyphs would get incorrect names—but the last two are quite essential, as the `msam10` and `msbm10` fonts incorrectly claim to have the coding scheme `TEX MATH SYMBOLS`.

```
\installfont{xma1000}{%
  yma1000 encoding mcin,%
  cmr10 encoding ot1upright,%
  cmmi10,%
  cmsy10 encoding omscal,%
  msam10 encoding msam,%
  msbm10 encoding msbm,%
  mccmhax,mccmkern,mcmmissing,%
  cmsy10-base,cmsy10-extra%
}{mc}{MC}{cm}{m}{n}{\10-}
```

Also note the explicit L^AT_EX size specification for the range “10 pt and up”.

The second example makes use of a suffix clause to combine the letters from one font with the digits from another.

```
\installfont{msbrj8t}{msbr8r,msbrc8r suffix /2,latin}{digit2,t1}
{T1}{msbj}{m}{n}{}
```

In this case, the glyph base contains the glyphs of Monotype Sabon (SabonMT)—under names such as `A` for ‘A’, `a` for ‘a’, and one for a lining digit one—as well

as the glyphs of Monotype Sabon Small Caps and Oldstyle Figures (SabonMT-SCOSF)—under names such as A/2 for ‘A’, a/2 for ‘A’, and one/2 for a hanging digit one. The `digit2.etx` file simply makes the definition

```
\setcommand\digit#1{#1/2}
```

which causes `t1.etx` to put zero/2 in slot 48 (digit zero), one/2 in slot 49 etc., instead of as it normally would zero in slot 48, one in slot 49 and so on. The net effect is that the digits in the generated `msbrj8t` is from `msbrc8r` (SabonMT-SCOSF) but everything else is from `msbr8r` (SabonMT).

The third example makes use of an `mtxasetx` clause to install (with its default encoding) a font for which creating an appropriate ETX file seems not worth the trouble.

```
\installrawfont{psyr}{psyr,\metrics
\setint{xheight}{\height{alpha}}
}{txtfdmns,psyr mtxasetx}{U}{psy}{m}{n}{}
```

The effect of the second `psyr` is that `psyr.mtx` is read (in case there was no `psyr.mtx` then it is created from (hopefully) `psyr.afm`) and the information in it will form the glyph base. Because of the `\metrics` control sequence, the rest of that item will be interpreted as explicit metric commands modifying the glyph base, and thus the `\setint` command can provide a value for the `xheight` variable (there doesn’t seem to be such a value in the AFM). Once the glyph base is completed, the `\installrawfont` starts writing the file `psyr.pl` (that’s for the first `psyr`). The encoding of that font will, because of the `psyr mtxasetx`, be the same as that used in `psyr.mtx`. Finally, the `txtfdmns` is for `txtfdmns.etx`, an ETX file which sets `fontdimens` 1–16 as for a T1 encoded font but does not set any slots. Since `psyr.mtx` reinterpreted as an ETX file sets slots but no `fontdimens`, these complement each other nicely.

10.7 Bounding boxes

Han The Thanh has created an implementation of bounding box support for `fontinst`, and it is a modified form of that support is distributed with `fontinst` as the file `bbox.sty`. To load this, begin your command file with

```
\input fontinst.sty
\input bbox.sty
```

The reason for not making it default is that keeping track of bounding boxes increases some of `fontinst`’s memory requirements quite a lot.

One important characteristic of this implementation is that the dimensions of the bounding box are not bundled into the same data structure (the `\g-⟨glyph⟩` macros) as the glyph’s width, height, depth, and italic correction are, but stored in a separate data structure (the `\gb-⟨glyph⟩` macros). A glyph doesn’t need to

have its bounding box set, it is simply a piece of information that fontinst will store if you tell it to and which you can later retrieve.

The bounding box will be stored as coordinates of the sides in the normal AFM coordinate system. The commands for retrieving these coordinates are

Command	Side
<code>\bbtop{<glyph>}</code>	top (<i>y</i> -coordinate)
<code>\bbbotttom{<glyph>}</code>	bottom (<i>y</i> -coordinate)
<code>\bbleft{<glyph>}</code>	left (<i>x</i> -coordinate)
<code>\bbright{<glyph>}</code>	right (<i>x</i> -coordinate)

In Thanh's implementation the command names were `\ury`, `\lly`, `\llx`, and `\urx` respectively instead, but I think the former are easier to remember. If no bounding box has been set for a glyph then the above commands will instead report the corresponding coordinate of the glyph's T_EX box (i.e. `\height{<glyph>}`, `\neg{\depth{<glyph>}}`, 0, and `\width{<glyph>}` respectively).

The command for setting the bounding box of a glyph is

```
\setglyphbb{<glyph>}{<left>}{<bottom>}{<right>}{<top>}
```

11 Miscellaneous notes

11.1 On verbatim, typewriter, and monowidth fonts

The verbatim, typewriter, and monowidth concepts are common sources of confusion for those who use fontinst to install fonts with L^AT_EX; in particular there are many misconceptions about the relation between them. The official view (of which not much has actually been brought forward) is that these concepts are really about three quite different things.

A font is a *monowidth* (monospaced, fixed-pitch) font if all glyphs in it have exactly the same width. Some font formats make special provisions for such fonts; the most notable example is the AFM format, where a single `CharWidth` keyword specifies the width for all glyphs in the font. Fontinst responds to this by including the command

```
\setint{monowidth}{1}
```

in the MTX file generated from an AFM, but that is everything that is hard-wired into the program. That a font is monowidth is however something that one should take note of when installing it for T_EX, as it means many of the glyphs in it have such a strange appearance that they are (pretty much) useless. The `endash` is for example usually only half as long as the `hyphen` and the letters in ligature glyphs are only half as wide as normal letters. Many of the ETX and MTX files that come with fontinst contain special commands to avoid making use of such degenerate glyphs.

That a font is a *typewriter* font really only means that it has a typewriterish look about it. The two most familiar typewriter fonts are probably Computer

cm`tt`: The quick brown fox jumps over the lazy dog.
cm`vtt`: The quick brown fox jumps over the lazy dog.

Figure 1: Two typewriter fonts

	TEX TEXT	TEX TEXT WITHOUT F-LIGATURES	TEX TYPEWRITER TEXT
non-italic	cm <code>b10</code>		
	cm <code>bx5-12</code>		
	cm <code>bxsl10</code>		
	cm <code>dunh10</code>		
	cm <code>ff10</code>		
	cm <code>fib8</code>		
	cm <code>r6-17</code>	cm <code>csc8-10</code> cm <code>r5</code>	cm <code>sltt10</code>
	cm <code>sl8-12</code>		cm <code>tcsc10</code>
	cm <code>ss8-17</code>		cm <code>tt8-12</code>
	cm <code>ssbx10</code>		
	cm <code>ssdc10</code>		
	cm <code>ssi8-17</code>		
	cm <code>ssq8</code>		
	cm <code>ssqi8</code>		
	cm <code>vtt10</code>		
italic	cm <code>bxti10</code>		
	cm <code>fi10</code>		cm <code>itt10</code>
	cm <code>ti7-12</code>		
	cm <code>u10</code>		

Table 1: “OT1-encoded” Computer Modern fonts, collected according to the actual font encoding

Modern Typewriter (cm`tt`) and Courier. Both of these fonts are monowidth, but there is no absolute rule about this. One of the standard \TeX fonts is for example Computer Modern Variable-width Typewriter (cm`vtt`), which is not a monowidth font, as Figure 1 shows.

The verbatim concept has very little to do with fonts at all; in \LaTeX it is considered to be a property of the environment (`verbatim`, `macrocode`, etc.) rather than a property of the font. The connection there is with fonts is that the encoding of the font must contain visible ASCII (as defined in Appendix C of *The \TeX book*) as a subset for the text to be rendered correctly. The cm`tt` family is the only one amongst the original Computer Modern fonts which meets this criterion and that is the primary grounds for the idea that these three concepts should be connected. Today that reason is at best a very weak one, as all T1-encoded fonts also meet the criterion of containing visible ASCII as a subset.

A circumstance which has probably added to the confusion is that OT1 is usually claimed to be an encoding. In reality the Computer Modern fonts that are declared in \LaTeX as being OT1 display as many as five different encodings, as shown in Table 1. Since most monowidth fonts are only used for setting ver-

batim text, there is some code in `ot1.etx` which automatically chooses a `TEX` `TYPEWRITER` `TEXT` encoding for the font when the `monowidth` integer is set. The only reason for this is the guess that this is what the user wanted.

Acknowledgements

I'd like to thank all of the fontinst α -testers, especially Karl Berry, Damian Cugley, Steve Grahthwohl, Yannis Haralambous, Alan Hoenig, Rob Hutchings, Constantin Kahn, Peter Busk Laursen, Ciarán Ó Duibhín, Hilmar Schlegel, Paul Thompson, Norman Walsh and John Wells, who made excellent bug-catchers!

Thanks to Barry Smith, Frank Mittelbach, and especially Sebastian Rahtz for many useful email discussions on how virtual fonts should interact with $\text{\LaTeX} 2_{\epsilon}$.

Thanks to Karl Berry and Damain Cugley for detailed comments on this documentation.

Thanks to David Carlisle for the use of his `trig` macros for calculating trigonometry.