1 Fonts

1.1 Introduction

This chapter will cover the details of defining fonts and collections of fonts, and it will explain how to go about installing fonts in both MkII and MkIV. It helps if you know what a font is, and are familiar with the ConTEXt font switching macros.

The original ConTEXt font model was based on plain TEX, but evolved into a more extensive one primarily aimed at consistently typesetting Pragma ADE's educational documents. The fact that pseudo caps had to be typeset in any font shape in the running text as well as superscripts, has clearly determined the design. The font model has been relatively stable since 1995.

Currently there are three layers of font definitions:

- simple font definitions: such definitions provide `\named` access to a specific font in a predefined size
- body font definitions: these result in a coherent set of fonts, often from a same type foundry or designer, that can be used intermixed as a 'style'
- typescript definitions: these package serif, sans serif, mono spaced and math and other styles in such a way that you can conveniently switch between different combinations

These three mechanisms are actually build on top of each other and all rely on a low level mapping mechanism that is responsible for resolving the real font file name and the specific font encoding used.

When TEX users install one of the TEX distributions, like TEX-live, automatically a lot of fonts will be installed on their system. Unfortunately it is not that easy to get a clear picture of what fonts are there and what is needed to use them. And although the `texmf` tree is prepared for commercial fonts, adding newly bought fonts is not trivial. To compensate this, ConTEXt MkII comes with `texfont.pl`, a program that can install fonts for you. And if the global setup is done correctly, MkIV and XETEX can use the fonts installed in your operation system without the need for extra installation work.

1.2 Font files and synonyms

In ConTEXt, whenever possible you should define symbolic names for fonts. The mapping from such symbolic names onto real font names can be done such that it takes place unnoticed for the user. This is good since the name depends on the encoding and therefore not seldom is obscure and hard to remember. The trick is knowing how to use the `\definefontsynonym` command.

The first argument is the synonym that is being defined or redefined. Redefinition is not only allowed but often very useful. The second argument is the replacement of the synonym. This replacement can be a real font name, but it can also be another synonym. The optional third argument can be used for to specify font settings.

```
\definefontsynonym [.1.] [.2.] [.3.]
                                OPTIONAL

1   TEXT

2   IDENTIFIER

3   encoding  =  IDENTIFIER
    features  =  IDENTIFIER
    handling  =  IDENTIFIER
    mapping   =  IDENTIFIER
```

There is no limit on the number of indirection levels, but the last one in the chain has to be a valid font name. ConTEXt knows it has reached the bottom level when there is no longer any replacement possible.

Font settings actually take place at the bottom level, since they are closely related to specific instances of fonts. Any settings that are defined higher up in the chain perculate down, unless they are already defined at the lower level.

`encoding`	The font file encoding for tfm-based (MkII) fonts.
`handling`	The font handling for MkII (see previous chapter).
`features`	The font handling for MkIV and XETEX (see previous chapter).
`mapping`	letter case change mapping for MkII that may be used in special cases; never actually used in the ConTEXt core. See chapter `??` on languages for details.

Here is an example of the use of font synonyms:

```
\definefontsynonym [Palatino] [uplr8t]   [encoding=ec]
```

In this example, the argumnet `uplr8t` is the real font (the actual file name is `uplr8t.tfm`, but file extensions are normally omitted), and it contains the metrics for the Type 1 font URW Palladio L in EC encoding. From now on, the name `Palatino` can be used in further font definitions to identify this font, instead of the dreadfully low–level (and hard to remember) name `uplr8t` and its accompanying encoding.

A note on font names: In pdfTEX, the real font is the name of the TEX metrics file, minus the extension, as we saw already. In XETEX and MkIV a font name is a bit more complex, because in both cases OpenType fonts can be accessed directly by their official font name (but with any embedded spaces stripped out) as well as via the disk file name.

In these two systems, ConTEXt first attempts to find the font using the official font name. If that doesn't work, then it tries to use the font by file name as a fallback. Since this is not very efficient and also because it may generate —harmless, but alarming looking— warnings it is possible to force ConTEXt into one or the other mode by using a prefix, so you will most often see synonym definitions like this:

```
\definefontsynonym [MSTimes]       [name:TimesNewRoman] [features=default]
\definefontsynonym [Iwona-Regular] [file:Iwona-Regular] [features=default]
```

In XeTeX, the `file` prefix implies that XeTeX will search for an OpenType font (with extension `otf` or `ttf`) and if that fails it will try to find a TeX font (with extension `tfm`). In MkIV, the list is a little longer: OpenType (`otf`, `ttf`), Type 1 (`afm`), Omega (`ofm`), and finally TeX (`tfm`).

The use of aliases to hide the complexity of true font names is already very useful, but ConTeXt goes further than that. An extra synonym level is normally defined that attaches this font name to a generic name like `Serif` or `Sans`.

```
\definefontsynonym [Serif] [Palatino]
```

An important advantage of using names like `Serif` in macro and style definitions is that it can easily be remapped onto a completely different font than `Palatino`. This is often useful when you are experimenting with a new environment file for a book or when you are writing a ConTeXt module.

In fact, inside an environment file it is useful to go even further and define new symbolic names that map onto `Serif`.

```
\definefontsynonym [TitleFont] [Serif]
```

By using symbolic names in the main document and in style and macro definitions, you can make them independent of a particular font and let them adapt automatically to the main document fonts. That is of course assuming these are indeed defined in terms of `Serif`, `Sans`, etcetera. All the ConTeXt predefined typescripts are set up this way, and you are very much encouraged to stick to the same logic for your own font definitions as well.

The list of 'standard' symbolic names is given in table 1.1

As mentioned earlier, the items in the third argument of `\definefontsynonym` perculate down the chain of synonyms. Occasionally, you may want to splice some settings into that chain, and that is where `\setupfontsynonym` comes in handy.

```
\setupfontsynonym [.⋮.] [.≡.]

1   IDENTIFIER
2   inherits from \definefontsynonym
```

For example, the predefined MkII typescripts for font handling that we saw in the previous chapter contain a sequence of commands like this:

```
\setupfontsynonym [Serif]       [handling=pure]
\setupfontsynonym [SerifBold]   [handling=pure]
\setupfontsynonym [SerifItalic] [handling=pure]
....
```

1.3 Simple font definitions

The most simple font definition takes place with `\definefont`.

name	style, alternative	explanation
Blackboard	--	Used by the \bbd macro
Calligraphic	--	Used by the \cal macro
Fraktur	--	Used by the \frak macro
Gothic	--	Used by the \goth macro
OldStyle	--	Used by the \os macro
MPtxtfont	--	The default font for MetaPost
Calligraphy	cg,tf	
Handwriting	hw,tf	
MathRoman(Bold)	mm,mr(bf)	
MathItalic(Bold)	mm,mi(bf)	
MathSymbol(Bold)	mm,sy(bf)	
MathExtension(Bold)	mm,ex(bf)	
MathAlpha(Bold)	mm,ma(bf)	
MathBeta(Bold)	mm,mb(bf)	
MathGamma(Bold)	mm,mc(bf)	
MathDelta(Bold)	mm,md(bf)	
Mono	tt,tf	
MonoBold	tt,bf	
MonoItalic	tt,it	
MonoBoldItalic	tt,bi	
MonoSlanted	tt,sl	
MonoBoldSlanted	tt,bs	
MonoCaps	tt,sc	
Sans	ss,tf	
SansBold	ss,bf	
SansItalic	ss,it	
SansBoldItalic	ss,bi	
SansSlanted	ss,sl	
SansBoldSlanted	ss,bs	
SansCaps	ss,sc	
Serif	rm,tf	
SerifBold	rm,bf	
SerifItalic	rm,it	
SerifBoldItalic	rm,bi	
SerifSlanted	rm,sl	
SerifBoldSlanted	rm,bs	
SerifCaps	rm,sc	

Table 1.1 Standard symbolic font names, and the style–alternative pair they belong to.

```
\definefont [.1.] [.2.] [.3.]
                         OPTIONAL
1   IDENTIFIER
2   FILE
3   TEXT
```

This macro defines a font with the same name as the first argument and you can use its name as an identifier to select that font. The second argument works in the same way as the second argument to `\definefontsynonym`: you can use either a font synonym or a real font. There is an optional third argument that can be either a bare number like `1.5` , or a named setup (see section ??). In case of a bare number, that is a local setting for the interline space. In case of a setup, that setup can do whatever it wants.

For instance:

```
\loadmapfile [koeieletters]
\definefont  [ContextLogo] [koeielogos at 72pt]
\ContextLogo \char 2
```

will result in

If you want a fixed size font like in the example above, you can define a font using the primitive TEX `at` or `scaled` modifiers.

Be warned that `at` is often useful, but `scaled` is somewhat unreliable since it scales the font related to its internal design size, and that is often unknown. Depending on the design size is especially dangerous when you use symbolic names, since different fonts have different design sizes, and designers differ in their ideas about what a design size is. Compare for instance the 10pt instance of a Computer Modern Roman with Lucida Bright (which more looks like a 12pt then).

```
\definefont [TitleFont] [Serif scaled 2400]
```

Hardcoded sizes can be useful in many situations, but they can be annoying when you want to define fonts in such a way that their definitions adapt themselves to their surroundings. That is why ConTEXt provides an additional way of scaling:

```
\definefont [TitleFont] [Serif sa 2.4]
```

The `sa` directive means as much as 'scaled at the body font size'. Therefore this definition will lead to a 24pt scaling when the (document) body font size equals 10pt. Because the definition has a lazy nature, the font size will adapt itself to the current body font size.

There is an extra benefit to using `sa` instead of `at`. Instead of a numeric multiplier, you can also use the identifiers that were defined in the body font environment that specified the related dimensions. For example, this scales the font to the `b` size, being 1.440 by default:

```
\definefont [TitleFont] [Serif sa b]
```

In fact, if you use a bare name like in

```
\definefont [TitleFont] [Serif]
```

it will internally be converted to

```
\definefont [TitleFont] [Serif sa *]
```

which in turn expands into the current actual font size, after the application of size corrections for super– and subscripts etc.

For example

```
\definefont [TitleFont] [Sans]
{\TitleFont test} and {\tfc \TitleFont test}
```

gives

test and test

A specialized alternative to `sa` that is sometimes useful is `mo`. Here the size maps onto to body font size only after it has passed through an optional size remapping. Such remappings are defined by the macro `\mapfontsize`:

```
\mapfontsize [.¹.] [.².]

1  DIMENSION
2  DIMENSION
```

Such remapping before applying scaling is sometimes handy for math fonts, where you may want to use slightly different sizes than the ones given in the body font environment. In the ConTEXt distribution, this happens only with the Math Times fonts, where the predefined typescript contains the following lines:

```
\mapfontsize [5pt]    [6.0pt]
\mapfontsize [6pt]    [6.8pt]
\mapfontsize [7pt]    [7.6pt]
\mapfontsize [8pt]    [8.4pt]
\mapfontsize [9pt]    [9.2pt]
\mapfontsize [10pt]   [10pt]
\mapfontsize [11pt]   [10.8pt]
\mapfontsize [12pt]   [11.6pt]
\mapfontsize [14.4pt] [13.2pt]
```

As we have seen, `\definefont` creates a macro name for a font switch. For ease of use, there is also a direct method to access a font:

```
\definedfont [.*.]

*   inherits from \definefont
```

Where the argument has exactly the same syntax as the second argument to `\definefont`. In fact, this macro executes `\definefont` internally, and then immediately switches to the defined font.

1.4 Defining body fonts

In older versions of ConTEXt, the model for defining fonts that will be described in this section was the top–level user interface. These days, typescripts are used at the top–level, and the body font definitions are wrapped inside of those.

Most commercial fonts have only one design size, and when you create a typescript for such fonts, you can simply reuse the predefined size definitions. Later on we will see that this means you can just refer to a `default` definition.

Still, you may need (or want) to know the details of body font definitions if you create your own typescripts, especially if the fonts are not all that standard. For example, because Latin Modern comes in design sizes, there was a need to associate a specific font with each bodyfont size. You may find yourself in a similar situation when you attempt to create a typescript for a 'professional' commercial font set.

The core of this intermediate model is the `\definebodyfont` command that is used as follows:

```
\definebodyfont [10pt] [rm] [tf=tir at 10pt]
```

This single line actually defines two font switches `\tf` for use after a `\rm` command, and `\rmtf` for direct access.

As one can expect, the first implementation of a font model in TEX is also determined and thereby complicated by the fact that the Computer Modern Roman fonts come in design sizes. As a result, definitions can look rather complex and because most TEX users start with those fonts, font definitions are considered to be complex.

Another complicating factor is that in order to typeset math, even more (font) definitions are needed. Add to that the fact that sometimes fonts with mixed encodings have to be used, i.e. with the glyphs positioned in different font slots, and you can understand why font handling in TEX is often qualified as 'the font mess'. Flexibility simply has its price.

Like most other TEX users, Hans Hagen started out using the Computer Modern Roman fonts. Since these fonts have specific design sizes, ConTEXt supports extremely accurate `\definebodyfont` definitions with specific font names and sizes for each combination. The following is an example of that:

```
\definebodyfont [12pt] [rm]
    [ tf=cmr12,
     tfa=cmr12 scaled \magstep1,
```

```
    tfb=cmr12 scaled \magstep2,
    tfc=cmr12 scaled \magstep3,
    tfd=cmr12 scaled \magstep4,
     bf=cmbx12,
     it=cmti12,
     sl=cmsl12,
     bi=cmbxti10 at 12pt,
     bs=cmbxsl10 at 12pt,
     sc=cmcsc10 at 12pt]
```

It should be clear to you that for fonts with design sizes, similar `\definebodyfont` commands will have to be written for each of the requested body font sizes. But many commercial fonts do not come in design sizes at all. In fact, many documents have a rather simple design and use only a couple of fonts for all sizes.

The previous example used the available TEX–specifications `scaled` and `at`, but (as we say already) ConTEXt supports special keyword that is a combination of both: `sa` (scaled at).

For example, for the Helvetica Type 1 font definition we could define:

```
\definebodyfont [12pt] [ss]
  [tf=hv  sa 1.000,
   bf=hvb sa 1.000,
   it=hvo sa 1.000,
   sl=hvo sa 1.000,
  tfa=hv  sa 1.200,
  tfb=hv  sa 1.440,
  tfc=hv  sa 1.728,
  tfd=hv  sa 2.074,
   sc=hv  sa 1.000]
```

The scaling is done in relation to the bodyfont size. In analogy with TEX's `\magstep` we can use `\magfactor`: instead of `sa 1.440` we could specify `sa \magfactor2`.

If you are happy with the relative sizes as defined in the body font environment (and there is no reason not to), the `\definebodyfont` can be four lines shorter. That is because ConTEXt predeclares a whole collection of names that combine the styles `rm`, `ss`, `tt`, `tf`, `hw` and `cg` with the alternatives `bf`, `it`, `sl`, `bi`, `bs`, and `sc` with the postfixes `a`, `b`, `c`, `d`, `x` and `xx`.

For the combination of `ss` and `sl`, the following identifiers are predeclared:

```
\ss    \ssa   \ssb   \ssc   \ssd    \ssx  \ssxx
\sl    \sla  \slb  \slc  \sld   \slx \slxx
\sssl  \sssla \ssslb \ssslc \sssld
```

And because there are no more sizes in the definition any more, we can just as well combine all of the requested sizes in a single `\definebodyfont` by using a list of sizes as the first argument. This means exactly the same as repeating that whole list five (or more) times, but saves a lot of typing:

```
\definebodyfont [12pt,11pt,10pt,9pt,8pt] [ss]
  [tf=hv  sa 1.000,
   bf=hvb sa 1.000,
   it=hvo sa 1.000,
   sl=hvo sa 1.000,
   sc=hv  sa 1.000]
```

Because the font names (may) depend on the encoding vector, we had better use the previously discussed method for mapping symbolic names. So, any one of the three following lines can be used, but the third one is best:

```
\definebodyfont [10pt,11pt,12pt] [ss] [tf=hv        sa 1.000]
\definebodyfont [10pt,11pt,12pt] [ss] [tf=Helvetica sa 1.000]
\definebodyfont [10pt,11pt,12pt] [ss] [tf=Sans      sa 1.000]
```

And in the actual ConTEXt core, the default body fonts are in fact defined with commands like this:

```
\definebodyfont [default] [rm]
  [ tf=Serif       sa 1,
    ...
    it=SerifItalic sa 1,
    ... ]
```

We saw that `\tf` is the default font. Here `\tf` is defined as `Serif sa 1` which means that it is a serif font, scaled to a normal font size. This `Serif` is mapped elsewhere on for example `Palatino` which in turn is mapped on the actual filename `uplr8t`, as demonstrated earlier.

```
\definebodyfont [...,1,...] [.,2,.] [..,.,3,.,..]
                                OPTIONAL
1   5pt ... 12pt small big
2   rm ss tt hw cg mm
3   tf  =  FILE
    bf  =  FILE
    sl  =  FILE
    it  =  FILE
    bs  =  FILE
    bi  =  FILE
    sc  =  FILE
    mr  =  FILE
    ex  =  FILE
    mi  =  FILE
    sy  =  FILE
    ma  =  FILE
    mb  =  FILE
    mc  =  FILE
    md  =  FILE
```

The macro syntax for \definebodyfont is a bit abbreviated. Besides the two–letter keys that are listed for the third argument, it is also possible to assign values to font identifiers with the alphabetic suffixes a through d like tfa as well as the ones with an x or xx suffix like bfx. You can even define totally new keywords, if you want that.

As an example we will define a bigger fontsize of \tf:

```
\definebodyfont [10pt,11pt,12pt] [rm]
    [tfe=Serif at 48pt,
     ite=SerifItalic at 48pt]
\tfe Big {\it Words}.
```

This becomes:

Big *Words*.

Note that there is a small trick here: the assignment to ite is needed for the command \it to work properly. Without that, the command \it would run the 'normal' version of it and that has a size of 11pt.

The keywords mr, ex, mi, sy, ma, mb,mc and md all relate to math families. As was already hinted at in table 1.1, these have extended relatives suffixed by bf for use within bold math environments.

Calls of \definebodyfont for the mm style look quite different from the other styles, because they set up these special keywords, and nothing else. The first four keys are required in all math setups just to do basic formula typesetting, the other four (ma . . . md) can be left undefined. Those are normally used for fonts with special symbols or alphabets like the AMS symbol fonts msam and msbm.

Here is what a setup for a fairly standard mm could look like:

```
\definebodyfont [10pt] [mm]
    [mr=cmr10,
     ex=cmex10,
     mi=cmmi10,
     sy=cmsy10]

\definebodyfont [17.3pt,14.4pt,12pt,11pt,10pt,9pt] [mm]
    [ma=msam10 sa 1,
     mb=msbm10 sa 1]
```

The keys mc and md are left undefined. This example explicitly shows how multiple \definebodyfonts are combined by ConTEXt automatically and that there is no need to do everything within a single definition (in fact this was already implied by the tfe trick above.)

Apart from the calling convention as given in the macro syntax that has already been shown, there are a few alternative forms of \definebodyfont that can be used to defined and call body fonts by name:

```
\definebodyfont [.1.] [.2.] [.3.]

1   IDENTIFIER
2   inherits from \setupbodyfont
3   inherits from \setupbodyfont
```

This was used in the default serif font defintion shown above: the first argument to \definebodyfont was the identifier default because these definitions were to be used from within other definitions.

An actual size will be provided by the commands at the top–level in the calling chain, the third argument in that \definebodyfont call will also be default instead of actually specifying settings.

```
\definebodyfont [.1.] [.2.] [.3.]

1   inherits from \setupbodyfont
2   inherits from \setupbodyfont
3   IDENTIFIER
```

The use of the default actually happens deep inside ConTEXt so there is clear code that can be shown, but if it was written out, a call would for example look like this:

```
\definebodyfont
  [17.3pt,14.4pt,12pt,11pt,10pt,9pt,8pt,7pt,6pt,5pt,4pt]
  [rm,ss,tt,mm]
  [default]
```

To end this section: for advanced TEX users there is the dimension–register \bodyfontsize. This variable can be used to set fontwidths. The number (rounded) points is available in \bodyfontpoints.

This way of defining fonts has been part of ConTEXt from the beginning, but as more complicated designs started to show up, we felt the need for a more versatile mechanism.

1.5 Typescripts and typefaces

On top of the existing traditional font module, ConTEXt now provides a more abstract layer of typescripts and building blocks for definitions and typefaces as font containers. The original

font definition files have been regrouped into such typescripts thereby reducing the number of files involved.

As we saw earlier, 'using' a typescript is done via the a call to the macro \usetypescript. Here is the macro syntax setup again:

```
\usetypescript [...,1...] [...,2...] [...,3...]
                           OPTIONAL      OPTIONAL
1   IDENTIFIER
2   IDENTIFIER
3   IDENTIFIER
```

Typescripts are in fact just organized definitions, and 'using' a typescript therefore actually means nothing more than executing the set of definitions that is contained within a particular typescript.

The main defining command for typescripts is a start–stop pair that wraps the actual macro definitions.

```
\starttypescript [...] [...] [...]
   ....
\stoptypescript
```

As with \usetypescript, there can be up to three arguments, and these two sets of arguments are linked to eachother: the values of the first and second argument in the call to \starttypescript of

```
\starttypescript [palatino] [texnansi,ec,qx,t5,default]
  ...
\stoptypescript
```

are what make the MkII-style call to \usetypescript

```
\usetypescript [palatino] [ec]
...
```

possible and meaningful: the first argument in both cases is the same so that this matches, and the second argument of \usetypescript appears in the list that is the second argument of \starttypescript, so this also matches. ConTEXt will execute all matching blocks it knows about: there may be more than one.

To perform the actual matching, ConTEXt scans through the list of known \starttypescript blocks for each of the combinations of items in the specified arguments of \usetypescript. These blocks can be preloaded definitions in TEX's memory, or they may come from a file.

There is a small list of typescript files that is tried always, and by using \usetypescriptfile you actually add extra ones at the end of this list.

The automatically loaded files for the three possible engines are, in first to last order:

pdftex	xetex	luatex	explanation
type-tmf	type-tmf	type-tmf	Core TEX community fonts
type-siz	type-siz	type-siz	Font size setups
type-one			Type 1 free fonts
	type-otf	type-otf	OpenType free fonts
	type-xtx		MacOSX font support
type-akb			Basic Adobe Type 1 mappings
type-loc	type-loc	type-loc	A user configuration file

Extra arguments to \usetypescript are ignored, and that is why that same two-argument call to \usetypescript works correctly in MkIV as well, even tough the typescript itself uses only a single argument:

```
\starttypescript [palatino]
  ...
\stoptypescript
```

On the other hand, extra arguments to \starttypescript are not ignored: a \starttypescript with two specified arguments will not be matched by a \usetypescript that has only one specified argument.

However, you can force any key at all to match by using the special keyword all in your \usetypescript or \starttypescript. We will see later that this use of a wildcard is sometimes handy.

1.5.1 A typescript in action

Before we can go on and explain how to write \starttypescript blocks, we have to step back for a moment to the macro \definetypeface, and especially to the third, fourth and fifth argument:

```
\starttypescript [palatino] [texnansi,ec,qx,t5,default]
\definetypeface[palatino] [rm] [serif] [palatino] [default]
...
```

Remember how in the previous chapter there were the tables that listed all the predefined combinations? It was said there that these '... are nothing more than convenience names that are attached to a group of fonts by the person that wrote the font definition'.

Here is how that works: these arguments of \definetypeface are actually used as parts of \usetypescript calls. To be preciese, inside the macro definition of \definetypeface, there are the following lines:

```
\def\definetypeface
  ...
  \usetypescript[#3,map][#4][name,default,\typefaceencoding,special]
  \usetypescript[#3][#5][size]
  ...
```

In our example #3 is serif, #4 is palatino, and #5 is default. The value of \typefaceencoding is inherited from the calling \usetypescript. That means that the two lines expand into:

```
\usetypescript[serif,map][palatino] [name,default,ec,special]
\usetypescript[serif][default][size]
```

And those typescripts will be searched for. This example is using MkII, so the list of typescript files is type-tmf, type-siz, type-one, type-akb, and type-loc. The first two arguments of \usetypescript are handled depth first, so first all 'serif' typescripts are tried against all the files in the list and then all the 'map' typescripts.

Not all of the seached typescript blocks are indeed present in the list of files that have to be scanned, but a few are, and one apparently even more than once:

```
type-tmf.tex  serif   palatino   name
type-one.tex  serif   palatino   texnansi,ec,8r,t5
type-one.tex  serif   palatino   ec,texnansi,8r
type-one.tex  map     all        –
type-siz.tex  serif   default    size
```

All of the found blocks are executed, so let's look at them in order

```
\starttypescript [serif] [palatino] [name]
    \definefontsynonym [Serif]            [Palatino]
    \definefontsynonym [SerifBold]        [Palatino-Bold]
    \definefontsynonym [SerifItalic]      [Palatino-Italic]
    \definefontsynonym [SerifSlanted]     [Palatino-Slanted]
    \definefontsynonym [SerifBoldItalic]  [Palatino-BoldItalic]
    \definefontsynonym [SerifBoldSlanted] [Palatino-BoldSlanted]
    \definefontsynonym [SerifCaps]        [Palatino-Caps]
\stoptypescript
```

This block has mapped the standard symbolic names to names in the 'Palatino' family, one of the standard font synonym actions as explained in the beginning of this chapter.

```
\starttypescript [serif] [palatino] [texnansi,ec,8r,t5]
\definefontsynonym [Palatino]
    [\typescriptthree-uplr8a]  [encoding=\typescriptthree]
\definefontsynonym [Palatino-Italic]
    [\typescriptthree-uplri8a] [encoding=\typescriptthree]
\definefontsynonym [Palatino-Bold]
    [\typescriptthree-uplb8a]  [encoding=\typescriptthree]
\definefontsynonym [Palatino-BoldItalic]
    [\typescriptthree-uplbi8a] [encoding=\typescriptthree]
\definefontsynonym [Palatino-Slanted]
    [\typescriptthree-uplr8a-slanted-167] [encoding=\typescriptthree]
\definefontsynonym [Palatino-BoldSlanted]
    [\typescriptthree-uplb8a-slanted-167] [encoding=\typescriptthree]
```

```
\definefontsynonym [Palatino-Caps]
    [\typescriptthree-uplr8a-capitalized-800] [encoding=\typescriptthree]

\loadmapfile[\typescriptthree-urw-palatino.map]
\stoptypescript
```

This maps the Palatino names onto the actual font files. Some further processing is taking place here: the calling `\usetypescript` that was called from within the `\definetypeface` knows that it wants `ec` encoding. Because this is the third argument, it becomes the replacement of `\typescriptthree`. The body of the typescript therefore reduces to:

```
\definefontsynonym[Palatino]            [ec-uplr8a]                [encoding=ec]
\definefontsynonym[Palatino-Italic]     [ec-uplri8a]               [encoding=ec]
\definefontsynonym[Palatino-Bold]       [ec-uplb8a]                [encoding=ec]
\definefontsynonym[Palatino-BoldItalic] [ec-uplbi8a]               [encoding=ec]
\definefontsynonym[Palatino-Slanted]    [ec-uplr8a-slanted-167]    [encoding=ec]
\definefontsynonym[Palatino-BoldSlanted][ec-uplb8a-slanted-167]    [encoding=ec]
\definefontsynonym[Palatino-Caps]       [ec-uplr8a-capitalized-800][encoding=ec]

\loadmapfile[ec-urw-palatino.map]
```

Incidentally, this also loads a font map file. In earlier versions of ConTEXt, this was done by separate typescripts in the file `type-map.tex`, but nowadays all map loading is combined with the definition of the synonyms that link to the true fonts on the harddisk. This way, there is a smaller chance of errors creeping in. See section 1.9 for more details on font map files.

The third match is a block that sets sets up 'TeXPalladioL' font synonyms. These will not actually be used, but it is a match so it will be executed anyway.

```
\starttypescript [serif] [palatino] [ec,texnansi,8r]
\definefontsynonym[TeXPalladioL-BoldItalicOsF]
    [\typescriptthree-fplbij8a][encoding=\typescriptthree]
...
\stoptypescript
```

The next matched entry loads the font map files for the default fonts:

```
\starttypescript [map] [all]
    \loadmapfile[original-base.map]
    \loadmapfile[original-ams-base.map]
\stoptypescript
```

this will not really be needed for the palatino `\rm` typescript, but it ensures that even if there is something horribly wrong with the used typescripts, at least pdfTEX will be able to find the Latin Modern (the default font set) on the harddisk.

The last match is the missing piece of the font setup:

```
\starttypescript [serif] [default] [size]
  \definebodyfont
    [4pt,5pt,6pt,7pt,8pt,9pt,10pt,11pt,12pt,14.4pt,17.3pt]
    [rm] [default]
\stoptypescript
```

and now the typescript is complete.

As explained earlier, that last block references a named `\definebodyfont` that is defined in `type-unk.tex`:

```
\definebodyfont [default] [rm]
  [tf=Serif sa 1,
   bf=SerifBold sa 1,
   it=SerifItalic sa 1,
   sl=SerifSlanted sa 1,
   bi=SerifBoldItalic sa 1,
   bs=SerifBoldSlanted sa 1,
   sc=SerifCaps sa 1]
```

similar `default` blocks are defined for the other five font styles also.

Looking back, you can see that the Palatino-specific typescripts did actually do anything except defininining font synonyms, loading a map file, and calling a predefined `bodyfont`.

1.5.2 Some more information

As we saw already, typescripts and its invocations have up to three specifiers. An invocation matches the script specification when the three arguments have common keywords, and the special keyword `all` is equivalent to any match.

Although any keyword is permitted in any of the three arguments, the current definitions (and macros like `\definetypeface`) make heavy use of some keys in particular:

pattern	application
`[serif] [*] [*]`	serif fonts
`[sans] [*] [*]`	sans serif fonts
`[mono] [*] [*]`	mono spaced fonts
`[math] [*] [*]`	math fonts
`[*] [*] [size]`	size specifications
`[*] [*] [name]`	symbolic name mapping
`[*] [*] [special]`	special settings
`[*] [all] [*]`	default case(s)
`[map] [*] [*]`	map file specifications

When you take a close look at the actual files in the distribution you will notice a quite a few other keywords. One in particular is worth mentioning: instead of the predefined sizes in `default`, you can use the `dtp` size scripts with their associated body font environments by using

```
\usetypescript [all] [dtp] [size]
```

or

```
\definetypeface[palatino] [rm] [serif] [palatino] [dtp]
```

In the top–level typescript for the palatino, we had a bunch of `\definetypeface` commands, as follows:

```
\definetypeface [funny] [rm] [serif] [palatino] [default] [encoding=texnansi]
\definetypeface [funny] [ss] [sans]  [palatino] [default] [encoding=texnansi]
\definetypeface [funny] [tt] [mono]  [palatino] [default] [encoding=texnansi]
\definetypeface [funny] [mm] [math]  [palatino] [default] [encoding=texnansi]
```

Once these commands are executed (wether or not as part of a typescript), `\funny` will enable this specific collection of fonts. In a similar way we can define a collection `\joke`.

```
\definetypeface [joke] [rm] [serif] [times]     [default] [encoding=texnansi]
\definetypeface [joke] [ss] [sans]  [helvetica] [default] [rscale=0.9,
                                                           encoding=texnansi]
\definetypeface [joke] [tt] [mono]  [courier]   [default] [rscale=1.1,
                                                           encoding=texnansi]
\definetypeface [joke] [mm] [math]  [times]     [default] [encoding=texnansi]
```

And the familiar Computer Modern Roman as `\whow`:

```
\definetypeface [whow] [rm] [serif] [modern] [latin-modern] [encoding=ec]
\definetypeface [whow] [ss] [sans]  [modern] [latin-modern] [encoding=ec]
\definetypeface [whow] [tt] [mono]  [modern] [latin-modern] [encoding=ec]
\definetypeface [whow] [mm] [math]  [modern] [latin-modern] [encoding=ec]
```

Now has become possible to switch between these three font collections at will. Here is a sample of some text and a little bit of math:

```
Who is {\it fond} of fonts?
Who claims that $t+e+x+t=m+a+t+h$?
Who {\ss can see} {\tt the difference} here?
```

When typeset in `\funny`, `\joke`, and `whow`, the samples look like:

Who is fond of fonts?
Who claims that $t + e + x + t = m + a + t + h$?
Who can see `the difference` here?

Who is fond of fonts?
Who claims that $t + e + x + t = m + a + t + h$?
Who can see `the difference` here?

Who is fond of fonts?
Who claims that $t + e + x + t = m + a + t + h$?
Who can see the difference here?

With `\showbodyfont` you can get an overview of this font.

[funny]												\mr : Ag
	\tf	\sc	\sl	\it	\bf	\bs	\bi	\tfx	\tfxx	\tfa	\tfb	\tfc	\tfd
\rm	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag
\ss	Ag	Ag	*Ag*	*Ag*	**Ag**	***Ag***	***Ag***	Ag	Ag	Ag	Ag	Ag	Ag
\tt	Ag	Ag	*Ag*	*Ag*	**Ag**	***Ag***	***Ag***	Ag	Ag	Ag	Ag	Ag	Ag

Figure 1.1 The `funny` typeface collection.

[joke]												\mr : Ag
	\tf	\sc	\sl	\it	\bf	\bs	\bi	\tfx	\tfxx	\tfa	\tfb	\tfc	\tfd
\rm	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag
\ss	Ag	Ag	*Ag*	*Ag*	**Ag**	***Ag***	***Ag***	Ag	Ag	Ag	Ag	Ag	Ag
\tt	Ag	Ag	*Ag*	*Ag*	**Ag**	***Ag***	***Ag***	Ag	Ag	Ag	Ag	Ag	Ag

Figure 1.2 The `joke` typeface collection.

[whow]												\mr : Ag
	\tf	\sc	\sl	\it	\bf	\bs	\bi	\tfx	\tfxx	\tfa	\tfb	\tfc	\tfd
\rm	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag
\ss	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag
\tt	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag	Ag

Figure 1.3 The `whow` typeface collection.

When defining the joke typeface collection, we used a scale directive. The next sample demonstrates the difference between the non scaled and the scaled alternatives.

Who is fond of fonts?
Who claims that t + e + x + t = m + a + t + h?
Who can see `the difference` here?

Who is fond of fonts?
Who claims that t + e + x + t = m + a + t + h?

Who can see `the difference` here?

It may not be immediately clear from the previous examples, but a big difference between using typeface definitions and the old method of redefining over and over again, is that the new method uses more resources. This is because each typeface gets its own name space assigned. As an intentional side effect, the symbolic names also follow the typeface. This means that for instance:

```
\definefont[MyBigFont][Serif sa 1.5] \MyBigFont A bit larger!
```

will adapt itself to the currently activated serif font shape, here `\funny`, `\joke` and `\whow`.

A bit larger!
A bit larger!
A bit larger!

1.5.3 A bit more about math

Math is kind of special in the sense that it has its own set of fonts, either or not related to the main text font. By default, a change in style, for instance bold, is applied to text only.

```
$        \sqrt{625} =     5\alpha$
$\bf     \sqrt{625} =     5\alpha$
$        \sqrt{625} = \bf 5\alpha$
$\bfmath \sqrt{625} =     5\alpha$
```

The difference between these four lines is as follows:

$\sqrt{625} = 5\alpha$
$\sqrt{625} = 5\alpha$
$\sqrt{625} = 5\alpha$
$\sqrt{625} = 5\alpha$

In order to get a bold α symbol, we need to define bold math fonts.[1] Assuming the font's typescripts support bold math, the most convenient way of doing this is the following:

```
\definetypeface [whow] [mm]
  [math,boldmath] [modern] [default] [encoding=texnansi]
```

Bold math looks like this:

$\sqrt{625} = 5\alpha$
$\sqrt{625} = 5\alpha$
$\sqrt{625} = 5\alpha$
$\sqrt{\mathbf{625}} = \mathbf{5}$

[1] Bold math is already prepared in the core modules, so normally one can do with less code

The definitions are given on the next page. Such definitions are normally collected in the project bound file, for instance called `typeface.tex`, that is then manually added to the list of typescript files:

```
\usetypescriptfile[typeface] % project scripts
```

It is also possible to avoid typescripts. When definitions are used only once, it makes sense to use a more direct method. We will illustrate this with a bit strange example.

Imagine that you want some math formulas to stand out, but that you don't have bold fonts. In that case you can for instance scale them. A rather direct method is the following.

```
\definebodyfont
  [funny]
  [12pt,11pt,10pt,9pt,8pt,7pt] [mm]
  [mrbf=MathRoman     mo 2,
   exbf=MathExtension mo 2,
   mibf=MathItalic    mo 2,
   sybf=MathSymbol    mo 2]
```

Our math sample will now look like:

$\sqrt{625} = 5\alpha$
$\sqrt{625} = 5\alpha$
$\sqrt{625} = 5\alpha$
$\sqrt{625} = 5\alpha$

We can also use an indirect method:

```
\definebodyfont
  [smallmath] [mm]
  [mrbf=MathRoman     mo .5,
   exbf=MathExtension mo .5,
   mibf=MathItalic    mo .5,
   sybf=MathSymbol    mo .5]

\definebodyfont
  [funny]
  [12pt,11pt,10pt,9pt,8pt,7pt]
  [mm] [smallmath]
```

This method is to be preferred when we have to define more typefaces since it saves keystrokes.

$\sqrt{625} = 5\alpha$
$\sqrt{625} = 5\alpha$
$\sqrt{625} = 5\alpha$
$\sqrt{625} = 5\alpha$

For efficiency reasons, the font definitions (when part of a typeface) are frozen the first time they are used. Until that moment definitions will adapt themselves to changes in for instance scaling and (mapped) names. Freezing definitions is normally no problem because typefaces are defined for a whole document and one can easily define more instances. When you redefine it, a frozen font is automatically unfrozen.

1.6 Predefined font, style and alternative keywords

Some of the internal commands are worth mentioning because they define keywords and you may want to add to the list.

Font size switching is done with keywords like `twelvepoint` and commands like `\twelvepoint` or `\xii`, which is comparable to the way it is done in plain TEX. These commands are defined with:

```
\definebodyfontswitch [fourteenpointfour] [14.4pt]
\definebodyfontswitch [twelvepoint]       [12pt]
\definebodyfontswitch [elevenpoint]       [11pt]
\definebodyfontswitch [tenpoint]          [10pt]
\definebodyfontswitch [ninepoint]         [9pt]
\definebodyfontswitch [eightpoint]        [8pt]
\definebodyfontswitch [sevenpoint]        [7pt]
\definebodyfontswitch [sixpoint]          [6pt]
\definebodyfontswitch [fivepoint]         [5pt]
\definebodyfontswitch [fourpoint]         [4pt]
\definebodyfontswitch [xii]  [12pt]
\definebodyfontswitch [xi]   [11pt]
\definebodyfontswitch [x]    [10pt]
\definebodyfontswitch [ix]   [9pt]
\definebodyfontswitch [viii] [8pt]
\definebodyfontswitch [vii]  [7pt]
\definebodyfontswitch [vi]   [6pt]
```

But be warned that `\xi` is later redefined as a greek symbol.

The keys in `\setupbodyfont` are defined in terms of:

```
\definefontstyle [rm,roman,serif,regular]    [rm]
\definefontstyle [ss,sansserif,sans,support] [ss]
\definefontstyle [tt,teletype,type,mono]     [tt]
\definefontstyle [hw,handwritten]            [hw]
\definefontstyle [cg,calligraphic]           [cg]
```

In many command setups we encounter the parameter `style`. In those situations we can specify a key. These keys are defined with `\definealternativestyle`. The third argument is only of importance in chapter and section titles, where, apart from `\cap`, we want to obey the font used there.

```
\definealternativestyle [mediaeval]          [\os]               []
\definealternativestyle [normal]             [\tf]               []
\definealternativestyle [bold]               [\bf]               []
\definealternativestyle [type]               [\tt]               []
\definealternativestyle [mono]               [\tt]               []
\definealternativestyle [slanted]            [\sl]               []
\definealternativestyle [italic]             [\it]               []
\definealternativestyle [boldslanted,
                         slantedbold]        [\bs]               []
\definealternativestyle [bolditalic,
                         italicbold]         [\bi]               []
\definealternativestyle [small,
                         smallnormal]        [\tfx]              []
\definealternativestyle [smallbold]          [\bfx]              []
\definealternativestyle [smalltype]          [\ttx]              []
\definealternativestyle [smallslanted]       [\slx]              []
\definealternativestyle [smallboldslanted,
                         smallslantedbold]   [\bsx]              []
\definealternativestyle [smallbolditalic,
                         smallitalicbold]    [\bix]              []
\definealternativestyle [sans,
                         sansserif]          [\ss]               []
\definealternativestyle [sansbold]           [\ss\bf]            []
\definealternativestyle [smallbodyfont]      [\setsmallbodyfont] []
\definealternativestyle [bigbodyfont]        [\setbigbodyfont]   []
\definealternativestyle [cap,
                         capital]            [\smallcapped]      [\smallcapped]
\definealternativestyle [smallcaps]          [\sc]               [\sc]
\definealternativestyle [WORD]               [\WORD]             [\WORD]
```

In section ?? we have already explained how *emphasizing* is defined. With oldstyle digits this is somewhat different. We cannot on the forehand in what font these can be found. By default we have the setup:

```
\definefontsynonym [OldStyle] [MathItalic]
```

As we see they are obtained from the same font as the math italic characters. The macro `\os` fetches the runtime setting by executing `\symbolicfont{OldStyle}`, which is just a low-level version of `\definedfont[OldStyle sa *]`. A few other macros behave just like that:

macro	**synonym**	**default value**
`\os`	OldStyle	MathItalic (lmmi10)
`\frak`	Fraktur	eufm10
`\goth`	Gothic	eufm10
`\cal`	Calligraphic	cmsy10 (lmsy10)
`\bbd`	Blackboard	msbm10

In addition to all the alrady mentioned commands there are others, for example macros for manipulating accents. These commands are discussed in the file `font-ini`. More information can also be found in the file `core-fnt` and specific gimmicks in the file `supp-fun`. So enjoy yourself.

1.7 Symbols and glyphs

Some day you may want to define your own symbols, if possible in such a way that they nicely adapt themselves to changes in style and size. A good example are the eurosymbols. You can take a look in `symb-eur.tex` to see how such a glyph is defined.

```
\definefontsynonym [EuroSerif]     [eurose]
\definefontsynonym [EuroSerifBold] [euroseb]
...
\definefontsynonym [EuroSans]      [eurosa]
\definefontsynonym [EuroSansBold]  [eurosab]
...
\definefontsynonym [EuroMono]      [euromo]
\definefontsynonym [EuroMonoBold]  [euromob]
```

Here we use the free Adobe euro fonts, but there are alternatives available. The symbol itself is defined as:

```
\definesymbol [euro] [\getglyph{Euro}{\char160}]
```

You may notice that we only use the first part of the symbolic name. ConTEXt will complete this name according to the current style. You can now access this symbol with `\symbol [euro]`

	\tf	\bf	\sl	\it	\bs	\bi
Serif	€	€	€	€	€	€
Sans	€	€	€	€	€	€
Mono	€	€	€	€	€	€

More details on defining symbols and symbol sets can be found in the documentation of the symbol modules.

1.8 Encodings

TODO: Add macro syntax definition blocks

Until now we assumed that an `a` will become an a during type setting. However, this is not always the case. Take for example ä or æ. This character is not available in every font and certainly not in the Computer Modern Typefaces. Often a combination of characters `\"a` or a command `\ae` will be used to produce such a character. In some situation TEX will combine

characters automatically, like in `fl` that is combined to fl and not fl. Another problem occurs in converting small print to capital print and vice versa.

Below you see an example of the `texnansi` mapping:

```
\startmapping[texnansi]
  \definecasemap 228 228 196  \definecasemap 196 228 196
  \definecasemap 235 235 203  \definecasemap 203 235 203
  \definecasemap 239 239 207  \definecasemap 207 239 207
  \definecasemap 246 246 214  \definecasemap 214 246 214
  \definecasemap 252 252 220  \definecasemap 220 252 220
  \definecasemap 255 255 159  \definecasemap 159 255 159
\stopmapping
```

This means so much as: in case of a capital the character with code 228 becomes character 228 and in case of small print the character becomes character 196.

These definitions can be found in `enco-ans`. In this file we can also see:

```
\startencoding[texnansi]
  \defineaccent " a 228
  \defineaccent " e 235
  \defineaccent " i 239
  \defineaccent " o 246
  \defineaccent " u 252
  \defineaccent " y 255
\stopencoding
```

and

```
\startencoding[texnansi]
  \definecharacter ae 230
  \definecharacter oe 156
  \definecharacter o  248
  \definecharacter AE 198
\stopencoding
```

As a result of the way accents are placed over characters we have to approach accented characters different from normal characters. There are two methods: TEX does the accenting itself *or* prebuild accentd glyphs are used. The definitions above take care of both methods. Other definitions are sometimes needed. In the documentation of the file `enco-ini` more information on this can be found.

1.9 Map files

TODO: This section is too informal

If you're already sick of reading about fonts, you probably don't want read this section. But alas, dvi post processors and pdfTEX will not work well if you don't provide them `map` files that tell them how to handle the files that contain the glyphs.

In its simplest form, a definition looks as follows:

```
usedname < texnansi.enc < realname.pfb
```

This means as much as: when you want to include a file that has the `tfm` file `usedname`, take the outline file `realname.pfb` and embed it with the `texnansi` encoding vector. Sometimes you need more complicated directives and you can leave that to the experts. We try to keep up with changes in the map file syntax, the names of fonts, encodings, locations in the TEX tree, etc. However, it remains a troublesome area.

It makes sense to take a look at the `cont-sys.rme` file to see what preferences make sense. If you want to speed up the typescript processing, say (in `cont-sys.tex`:

```
\preloadtypescripts
```

If you want to change the default encoding, you should add something:

```
\setupencoding [default=texnansi]
```

You can let ConTEXt load the map files for pdfTEX:

```
\autoloadmapfilestrue
```

The following lines will remove existing references to map files and load a few defaults.

```
\resetmapfiles
\loadmapfile[original-base.map]
\loadmapfile[original-ams-base.map]
\loadmapfile[original-public-lm.map]
```

As said, map files are a delicate matter.

1.10 Installing fonts

TODO: Document use of MkIVand XETEX and in particular OSFONTDIR

Most TEX distributions come with a couple of fonts, most noticeably the Computer Modern Roman typefaces. In order to use a font, TEX has to know its characteristics. These are defined in `tfm` and `vf` files. In addition to these files, on your system you can find a couple of more file types.

suffix	content

tfm	TEX specific font metric files that, in many cases, can be generated from afm files
vf	virtual font files, used for building glyph collections from other ones
afm	Adobe font metric files that are more limited than tfm files (especially for math fonts)
pfm	Windows specific font metric files, not used by TEX applications
pfb	files that contain the outline specification of the glyphs fonts, also called Type 1
enc	files with encoding vector specifications
map	files that specify how and what font files are to be included

On your disk (or cdrom) these files are organized in such a way that they can be located fast.[2] The directory structure normally is as follows:

```
texmf / fonts  / tfm    / vendor   / name / *.tfm
               / afm    / vendor   / name / *.afm
               / pfm    / vendor   / name / *.pfm
               / vf     / vendor   / name / *.vf
               / type1  / vendor   / name / *.pfb
      / pdftex / config /                   *.cfg
               / config /                   *.map
               / config / encoding /        *.enc
```

The texmf-local or even better texmf-fonts tree normally contains your own fonts, so that you don't have to reinstall them when you reinstall the main tree. The pdftex directory contains the files that pdfTEX needs in order to make decisions about the fonts to include. The enc files are often part of distributions, as is the configuration cfg file. When you install new fonts, you often also have to add or edit map files.

ConTEXt comes with a Perl script texfont.pl that you can use to install new fonts. Since its usage is covered by a separate manual, we limit ourselves to a short overview.

Say that you have just bought a new font. A close look at the files will reveal that you got at least a bunch of afm and pfb files and if you're lucky tfm files.

Installing such a font can be handled by this script. For this you need to know (or invent) the name of the font vendor, as well as the name of the font. The full set of command line switches is given below:[3]

switch	**meaning**
fontroot	texmf font root (automatically determined)
vendor	vendor name (first level directory)
collection	font collection (second level directory)
encoding	encoding vector (default: texnansi)
sourcepath	when installing, copy from this path
install	copy files from source to font tree

[2] If you have installed teTEX or fpTEX (possibly from the TEXlive cdrom) you will have many thousands of font files on your system.

[3] there are a couple of more switches described in the manual mtexfonts.

`makepath`	when needed, create the paths
`show`	run tex on `*.tex` afterwards

You seldom need to use them all. In any case it helps if you have a local path defined already. The next sequence does the trick:

```
texfont --ve=FontFun --co=FirstFont --en=texnansi --ma --in
```

This will generate the `tfm` files from the `afm` files, and copy them to the right place. The Type 1 files (`pfb`) will be copied too. The script also generates a `map` file. When this is done successfully, a TEX file is generated and processed that shows the font maps. If this file looks right, you can start using the fonts. The TEX file also show you how to define the fonts.

This script can also do a couple of more advanced tricks. Let us assume that we have bought (or downloaded) a new font package in the files `demofont.afm` and `demofont.pfb` which are available on the current (probably scratch) directory. First we make sure that this font is installed (in our case we use a copy of the public Iwona Regular):

```
texfont --ve=test --co=test --ma --in demofont
```

We can now say:

```
\loadmapfile[texnansi-test-test.map]
\definefontsynonym[DemoFont][texnansi-demofont]
\ruledhbox{\definedfont[DemoFont at 50pt]Interesting}
```

Interesting

From this font, we can derive a slanted alternative by saying:

```
texfont --ve=test --co=test --ma --in --sla=.167 demofont
```

The map file is automatically extended with the entry needed.

```
\definefontsynonym[DemoFont-Slanted][texnansi-demofont-slanted-167]
\ruledhbox{\definedfont[DemoFont-Slanted at 50pt]Interesting}
```

Interesting

We can also create a wider version:

```
texfont --ve=test --co=test --ma --in --ext=1.50 demofont
```

When you use the `--make` and `--install` switch, the directories are made, fonts installed, and entries appended to the map file if needed.

```
\definefontsynonym[DemoFont-Extended][texnansi-demofont-extended-1500]
\ruledhbox{\definedfont[DemoFont-Extended at 50pt]Interesting}
```

Instead of using pseudo caps in TEX by using `\kap`, you can also create a pseudo small caps font.

```
texfont --ve=test --co=test --ma --in --cap=0.75 demofont
```

This method is much more robust but at the cost of an extra font.

```
\definefontsynonym[DemoFont-Caps][texnansi-demofont-capitalized-750]
\ruledhbox{\definedfont[DemoFont-Caps at 50pt]Interesting}
```

switch	meaning
`extend=factor`	stretch the font to the given factor
`narrow=factor`	shrink the font to the given factor
`slant=factor`	create a slanted font
`caps=factor`	replace lowercase characters by small uppercase ones
`test`	use test/test as vendor/collection

When manipulating a font this way, you need to provide a file name. Instead of a factor you can give the keyword `default` or a `*`.

```
texfont --test --auto --caps=default demofont
```

The previous example runs create fonts with the rather verbose names:

```
demofont
demofont-slanted-167
demofont-extended-150
demofont-capitalized-750
```

This naming scheme makes it possible to use more instances without the risk of conflicts.

In the distribution you will find an example batch file `type-tmf.dat` which creates metrics for some free fonts for the encoding specified. When you create the default font metrics this way, preferably `texmf-fonts`, you have a minimal font system tuned for you prefered encoding without the risk for name clashes. When you also supply `--install`, the font outlines will be

copied from the main tree to the fonts tree, which sometimes is handy from the perspective of consistency.

1.11 Getting started

TODO: This section needs to be modernized

The way TEX searches for files (we're talking web2c now) is determined by the configuration file to which the `TEXMFCNF` environment variable points (the following examples are from my own system):

```
set TEXMFCNF=T:/TEXMF/WEB2C
```

When searching for files, a list of directories is used:

```
set TEXMF={$TEXMFFONTS,$TEXMFPROJECT,$TEXMFLOCAL,!!$TEXMFMAIN}
```

Here we've added a font path, which itself is set with:

```
set TEXMFMAIN=E:/TEX/TEXMF
set TEXMFLOCAL=E:/TEX/TEXMF-LOCAL
set TEXMFFONTS=E:/TEX/TEXMF-FONTS
```

Now you can generate metrics and map files. The batch file is searched for at the ConTEXt data path in the texmf tree or on the local path.

```
texfont --encoding=ec --batch type-tmf.dat
```

If you want to play with encoding, you can also generate more encodings, like `8r` or `texnansi`.

```
texfont --encoding=texnansi --batch type-tmf.dat
texfont --encoding=8r       --batch type-tmf.dat
```

After a while, there will be generated `tfm`, `vf`, and `map` files. If you let ConTEXt pass the map file directives to pdfTEX, you're ready now. Otherwise you need to add the names of the mapfiles to the file `pdftex.cfg`. You can best add them in front of the list, and, if you use ConTEXt exclusively, you can best remove the other ones.

As a test you can process the TEX files that are generated in the process. These also give you an idea of how well the encoding vectors match your expectations.

Now, the worst that can happen to you when you process your files, is that you get messages concerning unknown `tfm` files or reports on missing fonts when pdfTEX writes the file. In that case, make sure that you indeed *have* the right fonts (generated) and/or that the map files are loaded. As a last resort you can load all map files by saying:

```
\usetypescript [map] [all]
```

and take a look at the log file and see what is reported.

In due time we will provide font generation scripts for installation of other fonts as well as extend the typescript collection.

1.12 Remarks

It really makes sense to take a look at the font and type definition files (`font-*.tex` and `type-*.tex`). There are fallbacks defined, as well as generic definitions. Studying styles and manual source code may also teach you a few tricks.