Generating Type 1 Fonts from METAFONT Sources

Taco Hoekwater
Kluwer Academic Publishers
Dordrecht
taco.hoekwater@wkap.nl

Abstract

This article makes a comparison between bitmapped and vector fonts, and presents some of the problems I encountered when I tried to convert METAFONT sources into PostScript Type 1 fonts.

The second part of this article will focus more closely on some of the problems that I faced while trying to convert METAFONTS into PostScript Type 1 fonts, but first some explanation is in order as to why one might want to do this conversion, and precisely what this conversion entails. These topics are the subjects of the first couple of paragraphs.

What are METAFONT Fonts?

How characters are created

I'll assume the reader knows the following: every TEX distribution has a program called METAFONT, that compiles font sources more or less the same way that the TEX program compiles text sources. A major difference between the two programs is that METAFONT produces *device–dependent* output (called `pk` files), whereas TEX produces *device–independent* output (also known as `dvi` files).

Let's look into the font sources that METAFONT uses, and see what kind of information they contain. These are ordinary ASCII files just like TEX sources, so it was easy to insert a listing of one of these files. The file that contains the METAFONT logo font (`logo10.mf`) suits our purpose quite well, since it is a rather simple font that probably everybody has available:

```
font_size:= 10pt#  ;
ht#      := 6pt#   ;
xgap#    := 0.6pt# ;
u#       := 4/9pt# ;
s#       := 0      ;
o#       := 1/9pt# ;
px#      := 2/3pt# ;
input logo
bye
```

What do we see here? First there are a bunch of assignments (the lines that contain `:=`), then there is an `input` (this command functions the same way as TEX's `\input`, so it will start reading the file `logo.mf` next), and finally the last command is `bye`.

The file `logo.mf` contains the actual commands to create the characters. It helps to run METAFONT now to see what is going on. Just type the following to a system command prompt:

```
mf logo10
```

Probably you didn't have to worry about where the `logo10.mf` file is on your hard disk, since most METAFONT implementations can do recursive directory searches just like TEX.

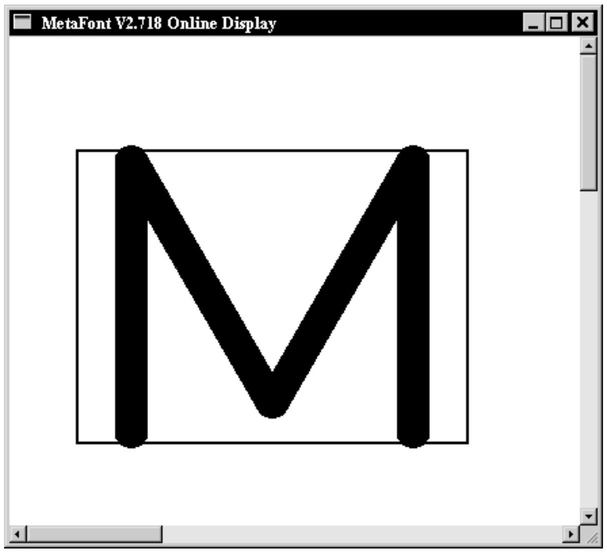


Figure 1 METAFONT output window for mode-less files

You should have seen a window popping up that shows characters as they are being created (like the one in figure 1, but it might look a little different on your system). Also, there should have been some terminal output, like this:

```
This is METAFONT, Version 2.718 (Web2c
7.2beta7) (logo10.mf (logo.mf [77] [69]
[84] [65] [70] [80] [83] [79] [78]) )
Output written on logo10.2602gf (9
characters, 98 80 bytes).
Transcript written on logo10.log.
```

The numbers you see are the positions of the characters in the font. For the METAFONT logo font, these are the positions of the used characters in the ASCII table: M, E, T, A, F, P, S, O, N. As you can see, they can appear in any order within the source files.

The file METAFONT has written is not precisely the same as the output on your screen, instead it looks like figure 2. Not really usable when it comes to typesetting text, but it contains some pretty valuable information nevertheless.

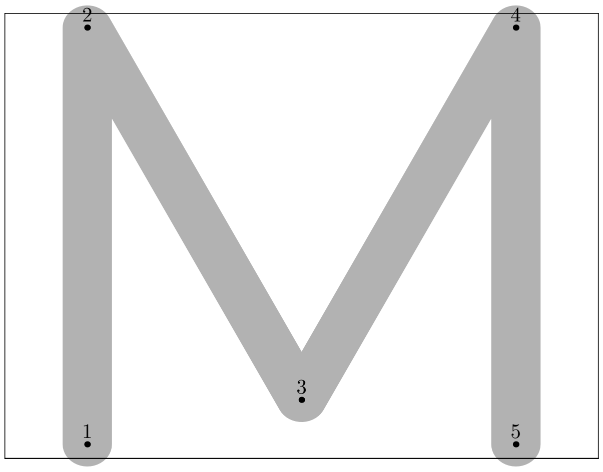


Figure 2 Metafont output file for modeless files

Have a look at the logo.mf file if you are interested in the nitty–gritty details. I will use only a small portion of that file to make some remarks about METAFONT. First, here is the edited program text I will use to explain things. (This is *no longer* valid METAFONT input, so don't start keying it in):

```
mode_setup;
define_pixels(s,u);

ygap#:=(ht#/13.5u#)*xgap#;
define_whole_pixels(xgap);
define_whole_vertical_pixels(ygap);

py#:=.9px#;
define_blacker_pixels(px,py);
pickup pencircle xscaled px yscaled py;
logo_pen:=savepen;
```

```
leftstemloc#:=2.5u#+s#;
define_good_x_pixels(leftstemloc);

beginlogochar("M",18);
x1 = x2 = leftstemloc;
x4 = x5 = w - x1;
x3      = w - x3;
y1      = y5;
y2      = y4;
bot y1  = 0;
top y2  = h;
y3      = ygap;
draw z1--z2--z3--z4--z5;
labels(1,2,3,4,5);
endchar;
```

Let us first look at the line that begins with beginlogochar, because this is where the real work is done. This portion of the source defines the letter 'M' in the font. What we see here is that characters are specified by first setting up a bunch of equations (the lines that have equal signs in them), followed by a draw command that connects those points, actually drawing the character.

We won't go deeply into METAFONT syntax, but it is vital to understand the following: a point is defined as a pair of x and y coordinates. In METAFONT syntax, points are sequentially numbered per character, starting from 1. z1 is the notation for point 1. Notations like x1 and y2 specify the x-component of point 1 and the y-component of point 2, respectively.

beginlogochar says that the 'M' is precisely $18u$ (units) wide, and logo10.mf has set up one u to be 4/9pt, so the actual character is $4/9 \times 18\text{pt} = 8\text{pt}$ wide.

TEX and METAFONT have the same author, and it shows: METAFONT can do macros just as easily as TEX can. Macros can have arguments, define other macros and assign values to things, just like in TEX. beginlogochar is in fact one of those macros, and it assigns some pretty important values when it gets expanded by METAFONT. For one, it defines w to be the width we calculated above, and it defines h to be the height of the character (calculated from ht# in logo10.mf, which equals 6pt).

From the values of u and s, it now follows that leftstemloc equals $(2.5 \times 4/9) + 0 = 10/9\text{pt}$. The last value we have left is ygap = (ht/13.5u)*xgap. ht and xgap have been given in the 'driver' file logo10.mf, and after some small calculations ygap becomes $(6\text{pt} \div (13.5 \times 4/9\text{pt} \times 0.6\text{pt})) = 0.6\text{pt}$.

It should now be easy to come to the conclusion that the equations fix the precise x, y locations of the five points that denote the character 'M', albeit in a slightly indirect manner. If we fill in the values we derived above, we get the following:

```
x1 = x2 = 10/9pt;
x4 = x5 = 8pt - x1;
x3      = 8pt - x3;
y1      = y5;
y2      = y4;
bot y1  = 0pt;
top y2  = 6pt;
y3      = 0.6pt;
```

After METAFONT has calculated these equalities for us, and with some minor reshuffling of the input, we get the following end-result:

```
x1 = 10/9pt  ; bot y1 = 0pt  ;
x2 = 10/9pt  ; top y2 = 6pt  ;
x3 = 4pt     ;     y3 = 0.6pt;
x4 = 62/9pt  ; top y4 = 6pt  ;
x5 = 62/9pt  ; bot y5 = 0pt  ;
```

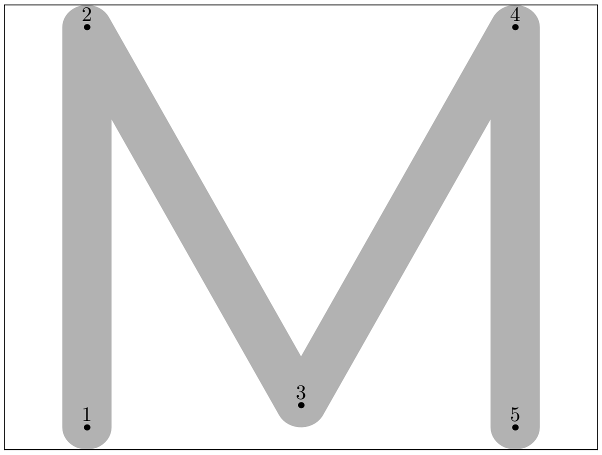


Figure 3 Metafont output file for modeless files, hand–calculated version

We could have typed this in right away, and METAFONT would have been just as happy. The end result would have been the same, as can be seen in figure 3.[1] But why do the calculation yourself if the machine can do it for you?

METAFONT's ability to do the needed calculations all by itself is one of its most important strong points. Combined with macros and separate input files, it becomes possible to use various fonts with the same shared sources. In such a 'font', the only file that is different between various versions of the font is the 'driver' file, that assigns different values to the same parameters. METAFONT's calculations will have different results, so that some of points will end up in slightly different locations. The resulting font will be similar in style but may still differ in lots of ways.

Creating a full font

Usually, fonts are not just a bunch of characters. There also is some other metric information included in almost every font. The final section of our example file (at the end, after all the characters have been defined) contains the following lines:

```
ligtable "T": "A" kern -.5u#;
ligtable "F": "O" kern -u#;
ligtable "P": "O" kern u#;

font_quad:=18u#+2s#;
font_normal_space:=6u#+2s#;
font_normal_stretch:=3u#;
font_normal_shrink:=2u#;
font_identifier:="MFLOGO" ;
font_coding_scheme:="AEFMNOPST only";
```

The first three lines belong to the 'ligature table'. Usually it will contain both real ligatures and the kerning information for the font, but because this is a very simple font, there are only three really simple kerning pairs.

The next lines define TEX's `\fontdimen` values: how wide a space will be and how much it can stretch and shrink, and some other information that will appear in the created font but is generally not used by programs.

Dealing with device dependence

Now let's have a look at the device dependent calculations that METAFONT does. Here is the relevant portion of the example again:

```
mode_setup;
define_pixels(s,u);

ygap#:=(ht#/13.5u#)*xgap#;
define_whole_pixels(xgap);
define_whole_vertical_pixels(ygap);
py#:=.9px#;
define_blacker_pixels(px,py);
```

There are, in fact, two kinds of device dependence that need to be dealt with. The `mode_setup` line takes care of the first kind of device dependence: the effects that the actual hardware of the printing engine can have on the printed font.

The most obvious difference between any two printing devices is of course the resolution, but there

1 Actually, figure 3 is not completely identical to figure 2, because in my example I cheated with the calculations a bit to keep the explanation simple.

are other problems as well. Since we prefer our output to look as close to our intended font as possible, usually a certain amount of correction is needed based on (i.e.) whether the device is going to be an inkjet printer or a laser typesetter.

mode_setup cannot do this all by itself, and this is why you usually have to specify somewhere what printer you are using. Programs like dvips will call METAFONT with a command like:

```
mf \mode=ljfour; mag=1; input logo10
```

If we forget about that first backslash, we can see that there are two assignments and one input command on this line. The second assignment differs from 1 when a font is called within TeX using a command like

```
\font\logohuge = logo10 at 20pt
```

In that case, the assignment would be mag=2. The other assignment is far more interesting. METAFONT usually starts with a 'format' file similar to the fmt files TeX uses, and somewhere in the sources for those format files there are some definitions like this:

```
mode_def cx =
  mode_param (pixels_per_inch, 300);
  mode_param (blacker, 0);
  mode_param (fillin, .2);
  mode_param (o_correction, .6);
  mode_common_setup_;
enddef;
```

Figure 4 An example of two different imaging models.

All parameters besides pixels_per_inch are a little too technical to explain in detail in a short article like this one, but figure 4 tries to explain that these values really do depend on the printing engine. The drawing on the left shows a more or less standard inkjet, that shoots dots of (black) ink on the paper. The right drawing shows a (hypothetical) printing device with a radically different approach. This machine pours light on a photographic film through a raster, creating a negative image. There are still round dots, but they are inverted! It is easy to imagine that this radically different technique can have quite an impact on the resulting image.

One effect that is very easy to see from the (admittedly very badly drawn) figure is that the inside corners in the right drawing are a lot blacker than in the left one. This sort of thing happens all the time in real life printing, but it often goes unnoticed because people tend to have only one printer.
The second device dependency is not really related to printers at all, but is caused simply by the fact that METAFONT outputs a pixel bitmap. Although METAFONT does its calculations with a very high accuracy, this does not help at all if there are simply not enough pixels to display the character. The commands that look like define_xxx_pixels take care of this kind of dependency, whose effects can be seen in figure 5.

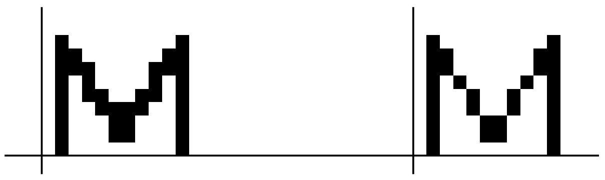


Figure 5 The character on the right has been created with all the define_xxx_pixels commands removed from the source.

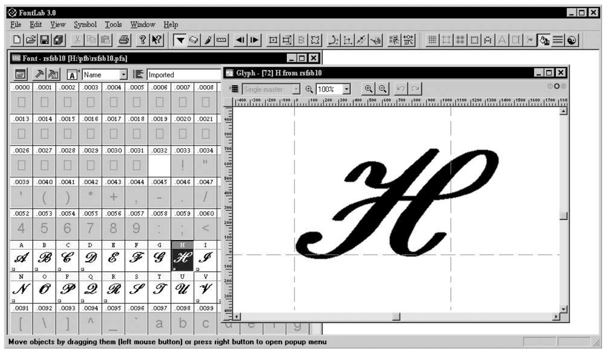
The sub-optimal distribution of pixels in this example is caused by the underlying pixel grid that can not be changed.

What are Type 1 fonts?

How PostScript fonts are created

PostScript Type 1 fonts are quite different from METAFONT fonts. Usually, Type 1 fonts are created in a wysiwyg environment with a drawing program that is only suited for the creation of fonts. Figure 6 shows the program I usually use.

The graphical user interface nicely shields the designer from what is happening behind the scenes, so we need to look into the generated files themselves if we want to get more information. On Windows and Unix systems, the actual fonts are saved in a binary file with the extension pfb (short for PostScript Font Binary), and the metric information in an ascii file with extension afm (short for Adobe Font Metrics).


FontLab 3.0
File Edit View Symbol Tools Window Help
Font - rsfsb10 [H:\pfb\rsfsb10.pfa]
Glyph - [72] H from rsfsb10

Move objects by dragging them (left mouse button) or press right button to open popup menu

Figure 6 An interactive font editor: Fontlab version 3

What a Type 1 font looks like[2]

The binary representation of a Type 1 font is just a compressed version of the non-compressed ascii format, with extension `pfa`. So we need a program that will do the decompression for us. One of the programs that can do this is `T1ascii` from the T1Utils package. But running this programs leaves us with a hexadecimal encrypted file. In the early days, the encryption key was a trade secret of Adobe Incorporated. This key is now freely available, but the file format still reflects the past. Yet another program from the T1Utils can convert this form to real human-readable PostScript: `T1disasm`. Now we can look at the generated PostScript file to see how the 'M' is defined in Type 1 format:

```
/M {
   78 800 hsbw
   611 -20 hstem
   -11 21 hstem
   0 66 vstem
   578 66 vstem
   581 595 rmoveto
   -259 -450 rlineto
   -259 450 rlineto
   -6 9 -12 7 -12 0 rrcurveto
   -16 -17 -12 -17 hvcurveto
   -563 vlineto
   -17 15 -13 18 vhcurveto
   19 14 13 17 hvcurveto
   439 vlineto
   76 -131 75 -131 75 -131 rrcurveto
   5 -10 12 -6 13 0 rrcurveto
   14 0 8 8 8 8 rrcurveto
   75 131 75 131 76 131 rrcurveto
   -439 vlineto
   -17 14 -13 19 vhcurveto
   18 15 13 17 hvcurveto
   562 vlineto
   17 -17 13 -16 vhcurveto
   -12 0 -12 -5 -6 -11 rrcurveto
   closepath
   endchar
} ND
```

The code looks enough like normal PostScript to recognize it at first glance, but the commands themselves are not the same ones you would use in everyday graphics. The PostScript language uses reverse Polish notation for its commands, so you should read backwards, starting at the end of the line. `581 595 rmoveto` means 'move to the point with coordinates $(581, 595)$.

All values are given in a coordinate system that maps 1000 units to one `em`. The nullpoint lies at the lower left corner. When one uses a PostScript font in a PostScript language program, the coordinate system is initially scaled in a way such that 1000 units equal precisely `1 bp`. The values used to describe points and intermediate values can be negative, but never partial. This need for discrete values can be a major problem when converting METAFONT fonts, as we will see later on.

Now let's have a short look at the used commands. The command `hsbw` sets up the width information for this character (the first number is the left sidebearing distance, the second number the advance width). The commands that end in `stem` are used by the hinting system. The whole collection of commands that look like `xlineto` and `xxcurveto` are shortcuts for the ordinary PostScript commands `lineto` and `curveto`: these draw the actual outline. All of these drawing commands are always relative to the 'current point'. The last couple of commands end the character: `closepath` to close the defined

2 This section is loosely borrowed from Erik-Jan Vens' article "Incorporating PostScript fonts in TEX", EuroTEX proceedings 1992, pp. 173–181.

path (like METAFONT's `cycle`) and `endchar` to do the actual drawing. `ND` functions as `def`: it defines the command '`M`' (from the first line) to mean 'do everything between the braces' (remember this is reverse Polish notation).

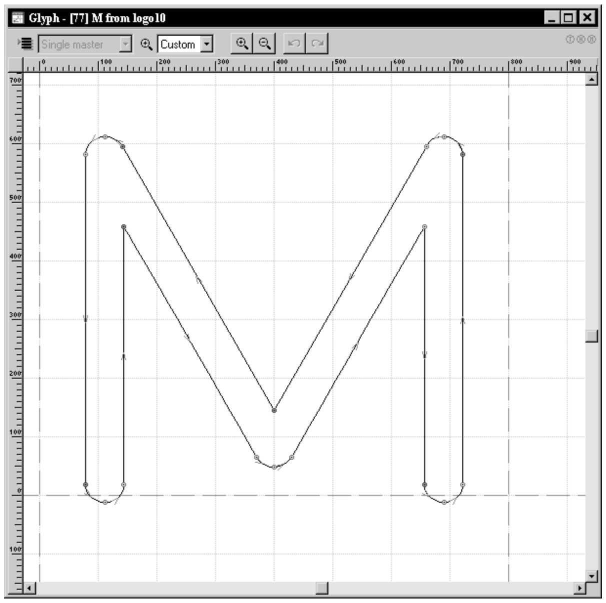


Figure 7 How the character's path is drawn.

At first sight it is a little surprising to see that the PostScript representation is rather a lot longer than the METAFONT version. This is caused by another limitation of Type 1 format: every character *has* to define an outlined path that is filled by `endchar`. Thanks to this limitation, we cannot use four stroked lines to draw the 'M' the way we did in METAFONT, but instead are forced to trace the borders of filled shape.

Dealing with device-dependencies

Adobe's Type 1 format does not supply a means of dealing with device differences directly, like METAFONT's `define_good_pixels`. But of course there has to be some means of making sure that a font looks reasonable on low-resolution devices, and this is handled by a system called 'hints'. The responsible commands are separated into two different levels: there are 'font-level' hints and 'character-level' hints. Font-level hints take care of three things:

1. Alignment zones
2. Standard stem widths
3. Extra information to control the hinting

The relevant portion of the font-file looks like this:

```
/BlueValues [ -12 0 600 611 ] ND
/BlueScale 0.04379 def
/BlueShift 7 def
/BlueFuzz 1 def
/MinFeature { 16 16 } ND
/StdHW [ 60 ] ND
/StdVW [ 66 ] ND
/ForceBold false def
```

Alignment zones. First off, alignment zones are defined by the array called `/BlueValues`. The values in the array define vertical zones by specifying two y coordinates for each zone. In this case, there are only the two areas between $[-12, 0]$ and $[600, 611]$, but there may be more entries.

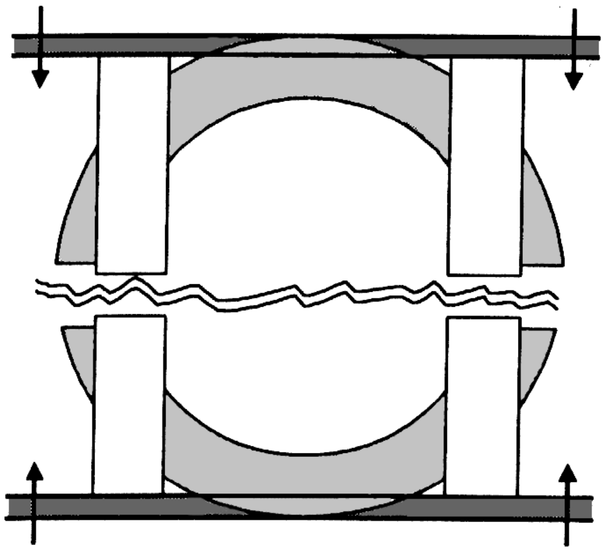


Figure 8 An example of overshoot-suppression in PostScript: the top and bottom of the 'O' are adjusted so that the character becomes just as high as the 'H'. (Figure borrowed from the Fontlab Manual.)

The first entry in the array defines an area in which the y-coordinates of points (that lie within this area) are changed into the highest (second) number. For the following entry, the y-coordinate is changed into the lowest (first) number. Together, these two areas allow characters like the 'O' to be rendered at low resolution without sticking out unacceptably below the baseline if compared to characters like the 'H' (see figure 8).

Standard stem widths. Quite often, a vertical or horizontal line in a font will be just a little bit too large for one device pixel but not large enough for two pixels. Depending on the underlying pixel grid, the line may consequently be rendered as either one

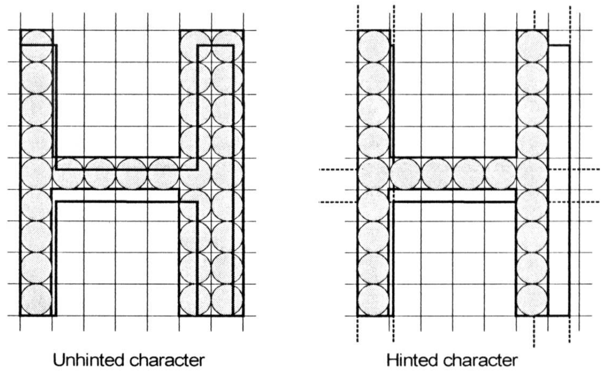


Figure 9 The desired result. (This figure is also borrowed from the Fontlab Manual.)

or two pixels.

In these problematic characters, we want to make sure at least that verticals and horizontals that are intended to have the same width throughout the font use the same number of pixels. This is done by pre-defining the widths that are supposed to be identical. The commands that pass this information to the renderer are `StdVW` and `StdHW`. The effect that a correct setting of these values has on the rendering of the font can be seen in figure 9 (Like all hinting information, these values are ignored if the stem widths are larger than three device pixels– approximating 1200 dpi for the average font. As a result, output at 1200 dpi on a new device sometimes looks inferior to the 600 dpi version for the trained eye.).

Extra information to control the hinting. There are some extra commands in the example that we haven't covered yet: the three `Bluexxxx` commands define (amongst other things) the pointsize below which overshoot suppression is turned on, and a fuzzy correction on the values of the alignment zones. `ForceBold` is used with bold fonts to make sure that they will stay at least two pixels wide at low resolutions (otherwise they would look identical to the non-bold version at small sizes).

The character-level hints are handled by the commands from the top of the listing given previously:

```
611 -20 hstem
-11 21 hstem
0 66 vstem
578 66 vstem
```

These define horizontal and vertical stem zones. The first number says at which coordinate to start, the second number the width to use from there. In this case (remember this is an 'M') there are two vertical stems, and two 'ghost' horizontal stems. Figure 10 shows the graphical representation of this character in the font editor.

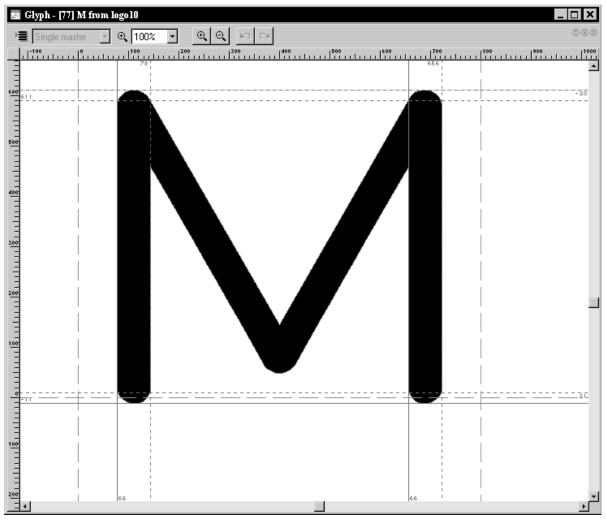


Figure 10 The character 'M' from the METAFONT logo font, with PostScript hints added

The 'ghost' stems are inserted because without them the overshoot suppression wouldn't work.

Why we want to convert to Type 1

Now that we have looked briefly into both formats, it is obvious that conversion from METAFONT input syntax to PostScript definitions is not going to be easy. METAFONT is apparently a lot smarter than the Type 1 interpreter, much better suited to handle device dependencies, and more accurate.

So, why bother at all? For practical reasons, of course. The most important incentive is the on-screen display of generated PDF files. Adobe's Reader is very bad at displaying bitmapped fonts, so files with only Type 1 fonts look a lot better. As it is, there are quite a few METAFONT fonts that don't (yet) have a Type 1 counterpart, so necessarily lots of TeX-generated PDF files use bitmaps.

There also is another interesting motive: designing high quality fonts in METAFONT syntax is a lot easier than creating Type 1 fonts of the same quality in an interactive editor (not to mention the fact that interactive programs usually crash at every second mouse click).

Tasks to be handled by the conversion process

Various things need to be taken care of by the conversion, but the three major parts are:

1. Resolving the equations in the METAFONT sources.
2. Converting stroked paths into outlined paths.
3. Insertion of Type 1 style hinting information.

Resolving the equations in the METAFONT sources

The first item is easy to do with an already existing program: METAPOST. METAPOST is a program by John Hobby (co-author of METAFONT) that accepts METAPOST input syntax and outputs an Encapsulated PostScript picture. For example, running METAPOST on the logo fonts (using precisely the same syntax as for METAFONT) gives the following output:

```
%!PS
%%BoundingBox: 0 -1 8 7
%%Creator: MetaPost
%%CreationDate: 1998.05.10:1535
%%Pages: 1
%%EndProlog
%%Page: 1 1
0.66418 0 dtransform exch truncate
     exch idtransform pop setlinewidth
[] 0 setdash
1 setlinecap
1 setlinejoin
10 setmiterlimit
gsave
newpath
1.10696 0.18819 moveto
1.10696 5.78938 lineto
3.98503 0.78595 lineto
6.8631 5.78938 lineto
6.8631 0.18819 lineto
1 0.9 scale
stroke
grestore
showpage
%%EOF
```

The PostScript code contained in this file is not that hard. The first few lines are just comments. The two lines that end with `setlinewidth` do nothing except setting the line width for strokes. It looks complex, but the code is always the same, the only things in these two lines that ever change are the two numbers.

The next lines set up some values of the PostScript graphics state that do not always have a predefined value (this is just a security measure). These lines also never change. `newpath` is the first command that is interesting: starting from here the character is defined. Indeed, there is only one `moveto`, followed by four straight lines, and finally a `stroke`.

It would be a little bit easier if the calculated values were given in units of a thousand per *em*, and this can be done by inserting a different `mode_def`. Basically, we ask METAPOST to generate a normal font file, but at a magnification of 100.375. This gives us an end result in PostScript big points, and the generated character will now look like this (some comments and irrelevant lines stripped):

```
%!PS
%%BoundingBox: 77 -12 723 612
66.66722 0 dtransform exch truncate
 exch idtransform pop setlinewidth
gsave newpath 111.1115 18.88889 moveto
111.1115 581.11142 lineto
399.99867 78.88953 lineto
688.88585 581.11142 lineto
688.88585 18.88889 lineto
1 0.90001 scale stroke grestore
showpage
%%EOF
```

Good. This is starting to look like something we could use. Ideally, we would prefer an output in rounded numbers, but that is not possible.

Converting stroked paths into outlined paths

Mr. Kinch (author of TrueTEX) has written a program called Metafog that converts METAPOST output as in the example above into the format required by the Type 1 specifications. At the moment, the program is only available as an optional extra with TrueTEX, but correspondence with Mr. Kinch indicate that it is very likely that this program will soon be available separately.

Metafog reads the METAPOST EPS file, and converts this into another EPS file. It also displays some debugging information about the character on the terminal:

```
interp: "scale" implies elliptical pen, \
                              66.7 x 6 0.0
o: scaffolded TRUE
reduce: reducing shape 1 of 2
reduce: reducing shape 2 of 2
```

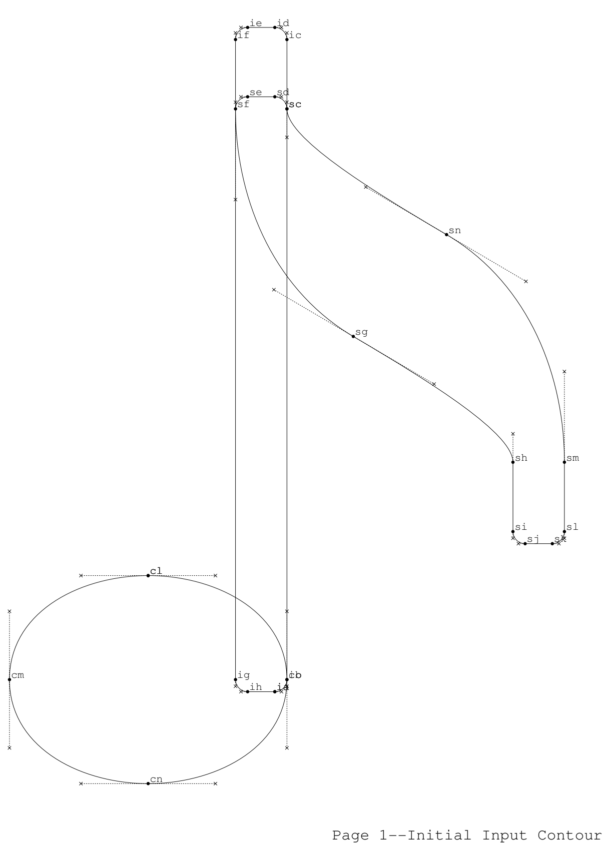


Figure 11 Metafog output file, page 1

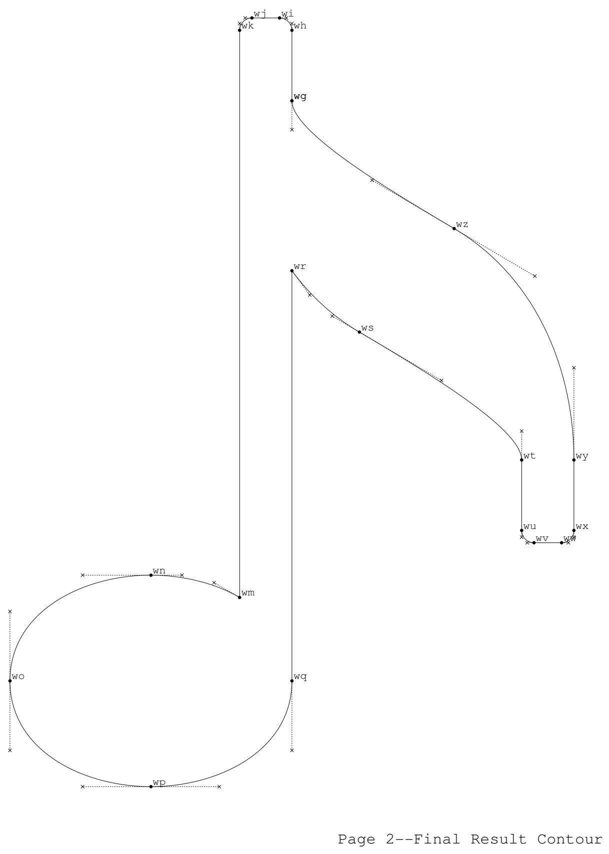


Figure 12 Metafog output file, page 2

```
duplicate: scaffolded
try_point: 0.01 value scaffold
reduce: reducing shape 3 of 2
.....
reduce: reducing shape 5 of 4
Plotting page 1 (Initial Input Contour) \
                         ... done plotting.
reduce: reducing shape 1 of 1
reduce: reducing shape 2 of 1
Plotting page 2 (Final Result Contour) \
                         ... done plotting.
Total knots used: 598 (a--wz), ~ 29% \
                         indexable capacity
```

The created files are pretty large, too large to include literally. This is because the file serves two purposes: it is the input format for another program (Makefont, also by Kinch) *and* it also shows the work that has been done by Metafog. The first page shows the result, the second page the initial input as Metafog saw it. (The output of one of these files is shown in figure 11 and figure 12.)

All we have to do is run Metafog on all characters in the font. If everything went correctly, the next step in the process is running the Makefont program. But this is not always the case. Metafog has its flaws, and it is especially bad at handling complex characters. One of those trickier characters is given in figure 13.

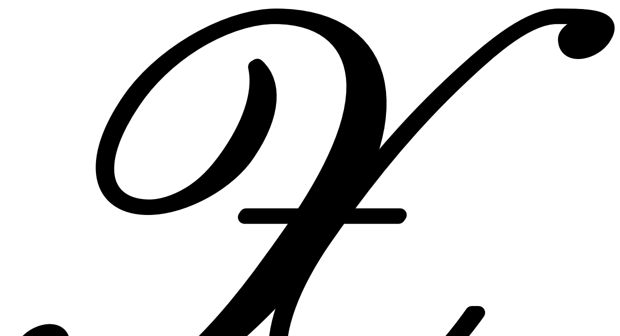


Figure 13 The character ‘X’ from Ralph Smith’s Formal Script

In order to handle cases like this gracefully, Metafog has a special startup option that gives a half-way result: it cuts the supplied shape in pieces, but it does not try to remove parts that are not needed.

There is yet another program in the Metafog suite that helps for the problematic characters.

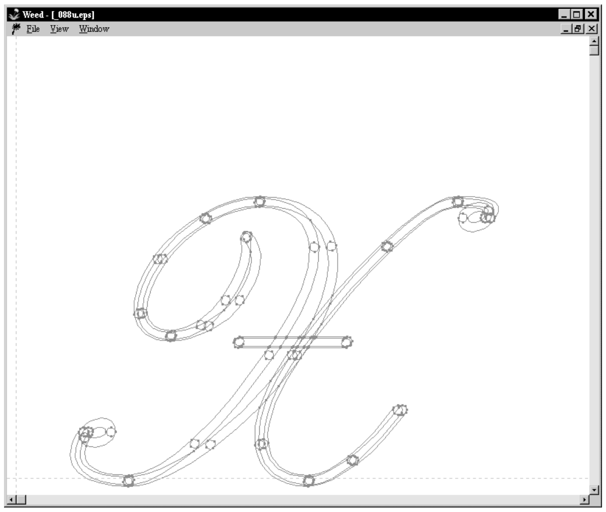


Figure 14 Screendump of the weeder's window

This program is called the 'weeder' (figure 14). It is an interactive program that reads the half-way result Metafog created. The human operator now has to select the partial lines that are supposed to belong to the shape, and the weeder will write a finished file for use by Makefont (just like Metafog itself would have done if things had gone right the first time).

For some fonts, one has to do almost every character 'by hand', for other fonts none at all. The point of view the machine has on what precisely denotes a complicated character can be rather unexpected: sometimes a character can look very simple to you but be almost impossible to process by Metafog (usually characters that use `draw` and `fill` commands that intersect somewhere). And the other way around also happens: large portions of the `nash14` (arabic) font looked exceedingly complex to me, but were in fact handled by Metafog without any problems.

Either way, eventually there will be EPS files available for all characters in the font. Makefont combines all of the separate files into one PostScript file, and the last step of the actual conversion process is running the T1Utils to get a binary representation that can be fed into a commercial font editor.

Insertion of Type 1 style hinting information

The `pfb` file created at the end of step two still has a couple of major flaws that need to be fixed. First and foremost among these: there are absolutely *no* Type 1 hints included. There *were* hints in the original METAFONT sources, but these are ignored by METAPOST, and subsequent portions of the conversion do not have access to them. This is the major reason for the need of a commercial font editor. Type 1 hinting is too complicated a process to rely on any non-interactive program to make the right choices.

Another thing that must be checked, especially for symbolic fonts, is the turning direction of the subpaths. In PostScript, whether a path will be black or white depends on how the path turns: clockwise or counterclockwise. Metafog sometimes gets confused, and outputs a character in which two concentric circles both turn leftward (like in an 'O' or the diameter symbol from the Waldi Symbol font). In those cases, the character will be completely filled, which is of course wrong.

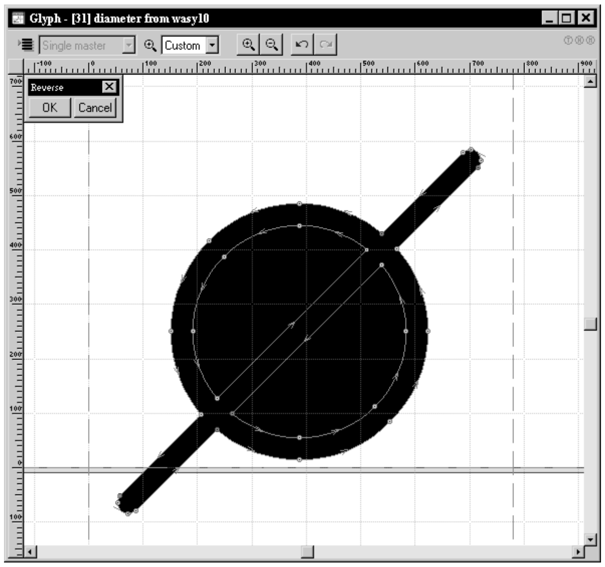


Figure 15 Paths that turn the wrong way

The absolutely final step (so far I've needed to do this for every commercial font program I could find) is disassembling the `pfb` file, running a perl script to fix some incompatibilies/bugs in the used font editor and to insert a couple of workarounds for bugs in software that uses the font, and reassembling.

What I have done already and future plans

So far, I have converted four METAFONT fonts that I needed myself. The files that are now available from CTAN are:

`logo` `wasy2` `stmary` `rsfs`

All of these files reside in a subdirectory of the METAFONT sources, named `ps-type1/hoekwater`. Each directory also contains a `README` file that gives some detailed information about the font in question and its copyrights.

Please take note of the fact that I *only* want to give support for problems that are *intrinsically* related to the `.pfb` files themselves. I don't have enough spare time to help people with problems related to the integration of the fonts into their TEX distributions. If somebody wants to volunteer for this job, please let me know and I will add you to the readme.

I plan to add other fonts in the near future (some of which may have been uploaded already by the time you read this). The original announcement of the availability of the files that are now on CTAN almost immediately resulted in a doubling of the length of my wish list.

The following fonts are *TODO* and will definitely be done before the summer:

- At least the most important fonts that are needed by the `wsuipa` package: `tipaxx` and `xipaxx`
- The Nash font that is used by ArabTEX (Klaus Lagally says that he needs to fix and update the METAFONT sources first).
- The Blackboard Bold font (actually, everything that is needed for the new math font encoding will be available in Type 1 before the end of the year).
- At least one each of the Greek, Cyrillic and Hebrew text font families (could someone please point me to the 'best' font of those that are available?).
- The `manfnt` (requested by Phil Taylor).

Every font (presuming 256 characters) takes about one day to complete. This does not sound like too much time, but unfortunately I also have other work to do :-(

I am still open for requests, but you may have to wait a couple of months.

For further reading

On the METAFONT *language:* Donald E. Knuth, "The METAFONTBook". Addison-Wesley Publishing Company, June 1986, *361 pages.*

On using METAFONT *to design real life fonts:* Donald E. Knuth, "Computer Modern Typefaces (Computers and Typesetting, volume E)". Addison-Wesley Publishing Company, June 1986, *588 pages.*

On the PostScript Language: Adobe Systems Inc, "The PostScript Language Reference Manual". Addison-Wesley Publishing Company, December 1990, *764 pages.*

On METAPOST*:* John Hobby, "A User's manual for METAPOST", AT&T Bell Laboratories Computing Science Technical Report 162, 1992. Comes as part of the METAPOST distribution.

On Type 1 fonts: Adobe Systems Inc, "Adobe Type 1 Font Format". Addison-Wesley Publishing Company, June 1995.

On the Metafog program: Richard J. Kinch, "Converting METAFONT Shapes to Outlines". Paper presented at the 1995 TUG Conference in St Petersburg, Florida, USA. Appeared in print in *TUGboat* 16.3