Special Fonts

Bogusław Jackowski* and Krzysztof Leszczyński†

abstract. We propose the use of a special pseudofont as an enhancement (in a sense) of the `\special` instruction. The examples of the implementation show that the technique applied here would prove to be extremely useful, especially with METAPOST.

keywords: cmdfont, special commands, MetaPost, fonts, PostScript

Keen users of TEX, METAFONT, or METAPOST might find the instructions called "special" very mighty helpers. However, METAPOST imposes serious limit on them: their content is placed at the very beginning of a PostScript file that METAPOST produces, just after the `%%Page` comment, before the very first real PostScript statement. It means that METAPOST, unlike TEX and METAFONT, is not able to intersperse drawing commands (`draw`, `fill`) or typesetting commands (`infont`, `btex ... etex`) with a special user-defined content. This behaviour embitters the life of METAPOST users and leads to neck-breaking solutions. Even worse, TEX `\special` instructions are ignored by dvitomp, making oodles of TEX packages unusable inside a `btex ... etex` construct.

The solution is to replace each `\special` instruction with a string typeset with a `special`-ly crafted font (pseudofont). We propose a natural name for it: `cmdfont`, *command font*. The text typeset with `cmdfont` has a special meaning when a TEX-generated dvi or a METAPOST-generated eps file is processed—it is treated as a sequence of commands to be interpreted.

The current article is the result of very preliminary thoughts—the idea is still very fresh. We are far from understanding all the consequences of this approach. Therefore, instead of developing a "general theory of specials," we decided to present just a few examples illustrating various possible applications of *special* `\special` *instructions*.

We have concentrated on using `cmdfont` with METAPOST. The use of pseudofonts with TEX, METAFONT, html or even major office editors is another thing. The reader may wish to evaluate the possibilities of special fonts. We perceive them as quite promising.

*`B.Jackowski@GUST.org.pl`

†Polish Linux Users' Group, `chris@linux.org.pl`

What is the special font?

Our special font can be defined with a short METAPOST program, `cmdfont.mp`:

```
designsize:=10bp/pt - epsilon;
fontdimen 2: designsize; % font normal space
fontmaking:=1;
for i:=0 upto 255:
 beginfig(i-256);
  charwd:=charht:=chardp:=charic:=0;
 endfig;
endfor
end.
```

The result of interpreting this program by METAPOST is `cmdfont.tfm`, i.e., a metric file which should be put somewhere where all TFM files reside. It contains 256 characters with all dimensions equal to zero. Most other font parameters are also set to zero. It is obvious that the font design size cannot be zero, but it is not obvious why the width of a space (`fontdimen2`) should be set to the design size. In fact, the actual size is not essential, any non-zero will do. Also, the actual design size value is not important as long as everybody who uses `cmdfont` takes *the same* `designsize` value.

Please note that the `beginfig` parameter is negative. Negative arguments instruct the METAPOST interpreter to output all the EPS files under the same name: `cmdfont.ps`. If we used a seemingly natural form, `beginfig(i)`, our directory would be infested by `cmdfont.0`, `cmdfont.1`, ..., `cmdfont.255` files. We don't need those files and it is easier to throw away one file than 256 files.

Special font in a METAPOST program

Let's trace how the instructions referring to `cmdfont` are parsed by METAPOST. Consider the file named, say, `infont.mp`:

```
beginfig(100);
 draw "META FONT" infont "cmdfont";
 draw "META POST" infont "cmdfont" scaled 2;
endfig;
end.
```

The resulting file, `infont.100`, reads:

```
1  %!PS
2  %%BoundingBox: 0 0 0 0
3  %%Creator: MetaPost
4  %%CreationDate: 2001.04.13:1950
5  %%Pages: 1
6  %*Font: cmdfont 10 10 20:80000000460708
7  %*Font: cmdfont 20 10 20:80000000440598
8  %%EndProlog
```

```
 9  %%Page: 1 1
10  0 0 moveto
11  (META FONT) cmdfont 10 fshow
12  0 0 moveto
13  (META POST) cmdfont 20 fshow
14  showpage
15  %%EOF
```

Leaving apart the hairy details of METAPOST-generated POSTSCRIPT code, let's note that the information about the font `cmdfont` is declared twice in the header of the file, namely, in the lines 6 and 7. Recall that METAPOST strings drawn by an `infont` command are always converted to a *single* `(Postscript string)`, even if they are extremely long. The space inside the METAPOST and the POSTSCRIPT string denotes the character of code 32. Computer Modern fonts use code 32 for the character *suppress* '‐' used for the letters 'Ł' and 'ł'. Most text fonts use code 32 for a normal non-stretching space.

Consider now a METAPOST program `btexetex.mp` that typesets a text using a construction `btex ... etex`:

```
verbatimtex \font\f cmdfont etex
beginfig(100);
 draw btex \f META FONT etex; draw btex \f META POST etex scaled 2;
endfig;
end.
```

It should not be a surprise that the resulting EPS differs from the previous one:

```
 1  %!PS
 2  %%BoundingBox: 0 0 0 0
 3  %%Creator: MetaPost
 4  %%CreationDate: 2001.04.13:1950
 5  %%Pages: 1
 6  %*Font: cmdfont 10 10 41:8c0e1
 7  %*Font: cmdfont 20 10 41:880b3
 8  %%EndProlog
 9  %%Page: 1 1
10  0 0 moveto
11  (META) cmdfont 10 fshow
12  9.9999 0 moveto
13  (FONT) cmdfont 10 fshow
14  0 0 moveto
15  (META) cmdfont 20 fshow
16  19.99979 0 moveto
17  (POST) cmdfont 20 fshow
18  showpage
19  %%EOF
```

The difference is that both strings have been split into two pieces (rows 13, 15 and 17, 19). Instead of typesetting a space character (`\char32`), TEX replaced each space with positioning instructions. Wizards able to read the magic hexadecimal sequences occurring in rows 6 and 7 will see that the character of code 32 is missing from the character set used to typeset texts in this particular PostScript file.

The problem of the space will recur in this article.

The special font and TEX+dvips

The files generated with METAPOST are usually included into TEX documents, calendered by dvips and eventually end up in the resulting PostScript file. The only font information TEX needs is the respective metric file (tfm). In contrast, dvips requires for a given font both the tfm file and its glyph shapes. It uses the header of the eps it processes to learn about the character set it needs. Unless we somehow monkey the header, dvips will demand that we provide a bitmap (pk) or a PostScript Type 1 (pfa or pfb) font file. But we have no glyphs for `cmdfont`—neither bitmaps, nor outlines.

We might have used the trick with a virtual font having all characters void, but it wouldn't work—dvips is smart enough to *ignore all texts* typeset with empty characters.

We have found no other way but to choose a popular font and identify it with `cmdfont` by adding an equivalence definition into the `psfonts.map` file. We have chosen `Courier`. The relevant line reads:

```
cmdfont Courier
```

That's all. Now the files typeset with `cmdfont` can be printed as if `cmdfont` was a regular font although the final effect might be weird. However, the `cmdfont` should be used in such a way that a PostScript interpreter would never attempt to display its characters.

How to use cmdfont without external processing

There are two PostScript instructions we have to bridle: `cmdfont` itself and `fshow`. METAPOST typesets the texts using the PostScript instructions with the name derived from the relevant tfm files. The `fshow` command is defined in the file `finclude.pro`. This file is automatically included when dvips encounters the eps file generated by METAPOST containing typeset texts. The piece of METAPOST code quoted below redefines the meaning of both instructions. Our `cmdfont` command interprets the string as an instruction sequence (`cvx exec`) and then it neutralizes the ensuing `fshow`.

```
def prep_cmdfont =
 special "/fshow where ";
 special " {pop} {/fshow {pop} def} ifelse";
 special "/cmdfont {cvx exec";
```

```
 special "  /fshow.tmp /fshow load def";
 special "  /fshow";
 special "    {pop /fshow /fshow.tmp load def}";
 special "   def";
 special "} def";
enddef;
extra_endfig:=extra_endfig & ";prep_cmdfont;";
```

Using the METAPOST `special` instruction guarantees that the code is moved to the begin of the POSTSCRIPT code and that's what we wanted to achieve. Note that splitting strings does not bother the '`cvx exec`' doublet.

Employing this technique yields undoubtedly useful results that are rather hard to achieve using "classic" methods. Below, we present three examples of feasible `cmdfont` applications. In the examples we refer to the file named `prepcmdf.mp` containing the definition of `prep_cmdfont` and `extra_endfig` assignment, as described above.

Example 1: Colouring fragments of a TEX text
Let's assume that the METAPOST illustration contains a text with a fragment to be coloured.

This is not a particularly
ingenious example of
colouring **a selected**
piece of text within
a `btex ... etex` clause.

The METAPOST source of this illustration is not particularly complicated:

```
input prepcmdf.mp;
verbatimtex
  \def\incmyk#1#2{%
    \leavevmode\rlap{\font\f=cmdfont \f
      gsave #1 setcmykcolor}%
      #2%
      {\font\f=cmdfont \f grestore}}
etex
beginfig(100);
 draw btex \vbox{
   \hsize 40mm \pretolerance10000 \raggedright \noindent
   This is not a particularly ingenious example of colouring
   \incmyk{0 0 0 0.3}{{\bf a selected piece of text}}
   within a~{\tt b{}tex} {\tt ...} {\tt e{}tex} clause.
 } etex scaled 1.2;
endfig;
```

The chief painter is the two-parameter macro `\incmyk` defined in the `verbatimtex ... etex` clause; the `\rlap` instruction used at the beginning of the macro definition

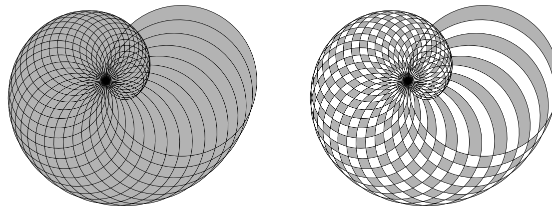
is crucial—we don't want the text "typeset" with `cmdfont` to influence the rest of the typesetting. Recall that the `cmdfont` space has a non-zero width.

This is the piece of the resulting EPS file responsible for the colour changes. The strings fed to `cmdfont` instructions are underlined to improve the legibility.

```
...
(gsave) cmdfont 11.99997 fshow
55.82301 28.69228 moveto
(0) cmdfont 11.99997 fshow
59.8076 28.69228 moveto
(0) cmdfont 11.99997 fshow
63.79219 28.69228 moveto
(0) cmdfont 11.99997 fshow
67.7768 28.69228 moveto
(0.3) cmdfont 11.99997 fshow
71.76138 28.69228 moveto
(setcmykcolor) cmdfont 11.99997 fshow
51.83855 28.69228 moveto
(a) plbx10 11.95514 fshow
62.50629 28.69228 moveto
(selected) plbx10 11.95514 fshow
0 14.3462 moveto
(piece) plbx10 11.95514 fshow
34.15456 14.3462 moveto
(of) plbx10 11.95514 fshow
49.21426 14.3462 moveto
(text) plbx10 11.95514 fshow
73.46477 14.3462 moveto
(grestore) cmdfont 11.99997 fshow
...
```

Example 2: The implementation of `eofill`

PostScript is armed with two basic countour-filling operations: `fill` and `eofill` (even-odd fill). The following picture illustrates the difference. "Snails" are constructed from the circles filled with `fill` (left side) and `eofill` (right side).



Unfortunately, METAPOST uses only `fill`. The `eofill` operator can be implemented using METAPOST `special` instructions. In general, however, it is hairy. Another

solution is the external processing of the resulting EPS files but this is even hairier. The use of `cmdfont` opens a rather simple way to implement `eofill`.

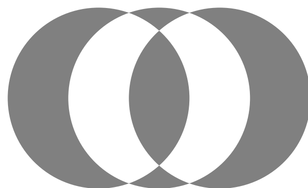
```
1  def eofill(text paths) text modif =
2   begingroup
3    save x_, y_;
4    for p_:=paths:
5     x_:=xpart(llcorner(p_));
6     y_:=ypart(llcorner(p_));
7     exitif true;
8    endfor
9    draw ("/fill.tmp /fill load def " &
10    "/newpath.tmp /newpath load def" &
11    "/fill {/fill{}def /newpath{}def}def")
12    infont "cmdfont" shifted (x_,y_) modif;
13   for p_:=paths: fill p_ modif; endfor
14   draw ("eofill /fill /fill.tmp load def " &
15    "/newpath /newpath.tmp load def")
16    infont "cmdfont"  shifted (x_,y_) modif;
17  endgroup
18 enddef;
```

Just a few words of comment: Lines 9–12 neutralize `fill` and `newpath` instructions appearing in the PostScript code generated by line 13. The code in lines 14–16 invokes `eofill` and restores the meaning of `fill` and `newpath`. `cmdfont` strings need to be positioned in a place that will not change the dimensions of the picture. In this example, strings are put in the lower left corner of the first path from the argument list (lines 4–8). This point satisfies our assumption: all strings have a total width of 0 because spaces are interpreted as characters of code 32.

The macro `eofill` can be used as follows:

```
eofill(
 fullcircle scaled 24mm shifted (-8mm,0),
 fullcircle scaled 24mm,
 fullcircle scaled 24mm shifted (8mm,0))
 withcolor 1/2white;
```

From the user's point of view, there is a difference between the `eofill` operation implemented here and the innate METAPOST `fill` operation—the argument of `eofill` is a list of paths rather than a single path. It is not difficult to predict the result of the code quoted above:



We do not present the PostScript code of the latter example just because boring the reader to death is not exactly our goal. Nevertheless, we recommend to run METAPOST and study the code, it is very instructive reading.

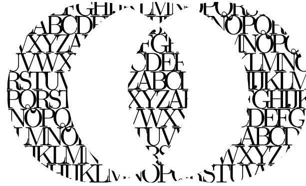
Example 3: The implementation of `eoclip`
The pair of `clip`-`eoclip` operators used for clipping the pictures is analogous to the `fill`-`eofill` pair we considered in the previous example. In particular, METAPOST provides only `clip`. The implementation of `\eoclip` using the `cmdfont` technique is a bit more difficult than that of `eofill`. Here's is our proposal:

```
1  def eoclip(expr pic)(text paths) text modif =
2   begingroup
3    save s_, xmin_, xmax_, ymin, ymax_;
4    xmin_=ymin_:=infinity; xmax_=ymax_:=-infinity;
5    draw ("/clip.tmp /clip load def " &
6     "/newpath.tmp /newpath load def" &
7     "/clip {/clip{}def /newpath{}def}def")
8     infont "cmdfont";
9    picture s_;
10   s_:=image(
11    draw ("eoclip /clip /clip.tmp load def " &
12    "/newpath /newpath.tmp load def")
13    infont "cmdfont"; draw pic);
14   for p_:=paths: clip s_ to p_ modif;
15    if xpart(llcorner(p_ modif)) < xmin_:
16     xmin_:=xpart(llcorner(p_ modif)); fi
17    if xpart(urcorner(p_ modif)) > xmax_:
18     xmax_:=xpart(urcorner(p_ modif)); fi
19    if ypart(llcorner(p_ modif)) < ymin_:
20     ymin_:=ypart(llcorner(p_ modif)); fi
21    if ypart(urcorner(p_ modif)) > ymax_:
22     ymax_:=ypart(urcorner(p_ modif)); fi
23   endfor
24   setbounds s_ to
25     (xmin_,ymin_)--(xmax_,ymin_)--
26     (xmax_,ymax_)--(xmin_,ymax_)--cycle;
27   addto currentpicture also s_;
28  endgroup
29 enddef;
```

The solution we propose is not obvious and has its drawbacks—we would gladly welcome any suggestions how to improve the code. Leaving apart the details, let's concentrate on a few things: (1) Part of the text typeset with `cmdfont` is added to the current picture (`currentpicture` variable, lines 5–8), another part is added to the local picture `s_` (lines 10–13). This is to ensure a proper order of PostScript operations. (2) The text is positioned at the coordinate origin, because eventually the

picture acquires its bounds explicitly (lines 24–26). (3) It resembles more the `eofill` operation defined previously than the original `clip`. We'll need to get used to it...

The illustration below presents the effect of `eoclip`.



It was generated by the following program, again admittedly trivial:

```
beginfig(100);
 picture p; p:=btex \vbox{
  \hsize 45mm \spaceskip-2ptplus1pt \parfillskip0pt
  \baselineskip7pt \lineskiplimit-\maxdimen \noindent
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
 } etex;
 eoclip(p)(
  fullcircle scaled 24mm shifted (-8mm,0),
  fullcircle scaled 24mm,
  fullcircle scaled 24mm shifted (8mm,0))
  shifted center p;
endfig;
```

The presented examples are supposed to convince the Reader that the use of `eofill` and `eoclip` is simple. Obviously, it does not imply that the respective definitions are simple. Nevertheless, we think that inventing such definitions is within every METAPOST user's reach. We count on it that METAPOST lovers will develop a heap of useful operations using the described techniques.

The special font and TeX packages

Processing the TeX fragments of a METAPOST source is one of the applications where the `cmdfont` technique proves useful. The basic METAPOST construction, i.e., `btex`

the TEX code etex works correctly if we stick to *classic* TEX only. Thus the phrase:

```
verbatimtex \input epsf etex;
picture P; P:=btex \epsfbox{file.eps} etex;
```

does not produce the desired results. The reason is the presence of TEX \special commands which (as we mentioned in the introduction) are ignored during the DVI-to-METAPOST transform. The macro \epsfbox, defined in the file epsf.tex from the DVIPS distribution, analyses the POSTSCRIPT file and generates the appropriate *whatsit* node of type *special*. For instance, the TEX file

```
\input epsf \epsfbox{tiger.ps} \end
```

where tiger.ps is a popular *on-duty* EPS file from the GHOSTSCRIPT distribution, produces \special{PSfile=tiger.ps llx=22 lly=171 urx=567 ury=738 rwi=5450}. If the TEX fragment is included in a METAPOST file, we should put \input epsf inside a verbatimtex ... etex block.

As we can see, we cannot move any further without knowing what to do with \special instructions. There should be a way to pass the information about them to METAPOST. One such way is redefining the \special instruction to transform its content to a string to be typeset with cmdfont. The naïve solution

```
\def \special #1{\smash{\hbox{\cmdfont TeXspecial #1}}}
```

does not work properly because the argument of \special usually contains spaces which have a non-zero width in cmdfont. TEX replaces such spaces by appropriate glues, thus splitting the argument into several substrings that are put into the final POSTSCRIPT file and interwoven with positioning instructions. Moreover, a \special argument may contain characters with unexpected categories (such as $). Let's assume that \special receives its argument as a string of characters of various categories but free from non-expandable non-character tokens (like \advance). In theory, such tokens may occur inside \special, but we haven't observed any single instance of such an instruction. (It does not mean, however, that they do not exist.) Our task is to convert a string into another string with all categories being "printable".

We propose the following trick: embed the argument of \special inside \csname ... \endcsname. This way, we get an "error trap" for free because \csname crashes when it finds a token that is not a character or a space. The resulting control sequence (with the meaning = \relax) can be converted by a \string command into a sequence of characters. As every TEX user knows, \string expands its arguments into a sequence of characters of category 12... well, not really—spaces get their "traditional" catcode: 10. Therefore all spaces must be converted into explicit \char32␣ characters by forcing TEX to write the character of code 32 into the DVI file. DVIPS or DVITOMP would convert such a character to an ordinary space. The simplified yet usable file cmdfont.tex redefining the \special command is given below:

```
\font \cmdfont=cmdfont
% The box keeps the string of characters used instead of \special
\newbox \mpspecialbox
```

```
% We'll keep all specials occurring in the main vertical list
% into the special box.
\newbox \MVLspecialbox
\setbox \MVLspecialbox=\null
\def \mpspecial #1{%
   \setbox \mpspecialbox=\hbox{%
      \cmdfont
      % set escapechar, just in case
      \escapechar=`\\%
% Prepare a macro with the name being the content of our special
% including a backslash at the very beginning.
      \edef \a {\expandafter \string
         \csname TeXspecial: #1\endcsname
         \space \relax}%
      % \b eats the leading backslash, that resulted from
      % \a after it was expanded.
      \def \b ##1{\c}
      % Change every space into the character coded 32.
      \def \c ##1 ##2\relax{##1%
         \ifx $##2$%
         \else \char32 \c##2\relax
         \fi}%
      \expandafter \b \a
   }%

   % If we're in the main vertical list, put the special into
   % a special box, otherwise just typeset it.
   \if \ifvmode\ifinner+\else-\fi\else+\fi +%
       \box \mpspecialbox
   \else
       \global \setbox \MVLspecialbox
            \hbox{\box \MVLspecialbox
                  \kern1sp
                  \box \mpspecialbox}%
   \fi
   }

\def\special{\mpspecial}
```

Such a redefinition of `\special` guarantees that its argument will not be ignored (by DVITOMP) and that METAPOST will receive their string equivalents. More importantly, every such `\special` generates a single string, therefore DVIPS will also generate a single string even if it is enormously large.

What should we do with such strings passed to METAPOST? The final POSTSCRIPT must be postprocessed with a SED, AWK, or PERL script. This processing is easier than

it could be, because, as we have mentioned, every special is transformed into a single `(Postscript string)`.

PostScript file postprocessing

A small example: `tiger.mp`

```
verbatimtex
  \input cmdfont
  \input epsf
etex

beginfig(100)
  draw btex \epsfxsize=20pt
        \epsfbox{tiger.ps} etex
endfig;
end.
```

The resulting file `tiger.100` contains:

```
%!PS
%%BoundingBox: 0 0 20 21
%%Creator: MetaPost
%%CreationDate: 2001.04.04:1968
%%Pages: 1
%*Font: cmdfont 10 10 20:800277e4000098805748bdc
%%EndProlog
%%Page: 1 1
9.9626 0 moveto
(TeXspecial: PSfile=tiger.ps llx=22 lly=171 u\
rx=567 ury=738 rwi=199) cmdfont
 10 fshow
showpage
%%EOF
```

The parameters `llx`, `lly`, `urx`, and `ury` define the bounding box of the figure to be included; `rwi` is equal to `\epsfxsize` $\times$ 10/`1bp`. We have to replace the string with the code that would be generated by DVIPS if indeed DVIPS found the relevant "real" `\special`.

We have to watch out for strings that begin with `(TeXspecial` and process them up to the final `) cmdfont ... fshow`.

What dvips understands

Although the popular DVI-to-PostScript translator called DVIPS understands lots of `\special` patterns, it does not accept all imaginable ones. It can tell friend from foe

by a `\special`'s prefix. Here is the list of the most frequently used prefixes that DVIPS can understand.

- ◇ `papersize`—Defines the page size; METAPOST deals with encapsulated PostScript files, thus we can ignore this parameter in most documents as setting page parameters is not allowed in EPS files (according to the Adobe specification of PostScript).
- ◇ `landscape`—Specifies page orientation; can be ignored, too.
- ◇ `header`—Adds the specified file to the header of the PostScript file made by DVIPS. Keeping in mind that METAPOST files are usually included in TeX documents and processed by DVIPS, we can *clone* this instruction by adding it to an auxiliary TeX file as a *normal* `\special`. If DVIPS felt moved by the standard PostScript structured comment `%%DocumentNeededResources`, we could replace the header `\special` by the METAPOST `special`. Actually, DVIPS would just ignore such a comment stolidly.
- ◇ `psfile`—Adds an EPS file. Unfortunately, there's no other way except adding such a file by hand or rather by script. Some non-standard elements like `%*Font` declarations could be cloned to the auxiliary file TeX file.
- ◇ `!`—Its argument is a piece of a literal PostScript code. Normally, DVIPS places it in the header of the final output file. Thus, we should clone it to the auxiliary TeX file.
- ◇ `"`—A piece of PostScript code, embedded in the graphic environment (coordinate transformation matrix) of an EPS file.
- ◇ `ps:` (note the single colon)—A piece of PostScript code embedded in the graphic environment (coordinate transformation matrix) of the PostScript file generated by DVIPS. Before and after the code DVIPS adds the positioning instructions.
- ◇ `ps::`, `ps::[begin]`, `ps::[end]`—These constructions are used to concatenate a sequence of `\special` instructions; They are omens of serious trouble. Their description in the DVIPS manual is rather laconic. One can really Ƨꟼoil things. We'll assume there are no such specials in the processed files.
- ◇ `em:`—This form was introduced by Eberhard Mattes and used to be understood only by `em`TeX. Although they were quite useful at the time, now they are mostly replaced by pure PostScript code. We won't deal with them.
- ◇ `html:`— ... well, perhaps some other time `:-)`.

Construction cloning

Assuming that every METAPOST file eventually gets into TeX and DVIPS, we can save our labour and many opportunities of making errors if we clone some constructions. A `\special` instruction that begins with a `header` or an exclamation mark prefix can be written, as was mentioned, to an auxiliary TeX file to be `\input` again in the final stage of processing. The structured `%*Font` comments can be saved to a pseudo-EPS

file containing only these comments; moreover, an appropriate `\special` command (`\special{psfile ...}`) can be added to the auxiliary TEX file. In this way, the relevant fonts will be included by DVIPS.

The best method to gain some insight into the techniques described herein is to experiment. One can process METAPOST files using the program `despecials` available from `ftp://bop.eps.gda.pl/pub/cmdfont`. It is a PERL script that updates the METAPOST output. It changes `\special` command equivalents expressed by `cmdfont` strings into proper (or sometimes improper) POSTSCRIPT code. Additionally, it produces the TEX file `mpspec.inc` containing selected cloned `\special` commands.

Header files

Life becomes worse if a TEX file that we add using `btex ... etex` generates `\special` instructions during input. Many macro libraries behave in such a way. As an illustration, let's take a very simple example of the output produced by the LILYPOND program for typesetting music. One of the possible effects of LILYPOND processing its score file is a TEX file (`gamac.tex` in this case) that can be input into the METAPOST figure. Even such a simple example contains four `\special` instructions after conversion to TEX format: two of them are placed in the header part and two are required for slurs. Typesetting scores is, no doubt, one of the most intricate tasks TEX can carry out and there is no way to do it without being *special-infested*. A one-page minuet from the LILYPOND manual contains more than 60 `\special` commands.

Our METAPOST file would look like:

```
verbatimtex
 \input cmdfont
 \input lily-ps-defs
etex
beginfig(100)
 picture P;
 P=btex \input gamac.tex etex;
 draw P;
endfig;
end.
```

Unfortunately, macros contained in the file `lily-ps-defs.tex` generate `\special` commands themselves. Let's consider the result of processing such a file by METAPOST: it generates a file `mpx$$.tex` (the exact name varies from system to system) with an obvious content:

```
\input cmdfont
\input lily-ps-defs
%
\shipout\hbox{\smash{\hbox{\hbox{%
```

```
\input gamac.tex}\vrule width1sp}}}
\end{document}
```

We can see two lines issued by `verbatimtex ... etex`, and the content of the `btex ... etex` block embedded in a rather complicated Russian doll of boxes, to be sent to the DVI file by an explicit `\shipout` command.

Without additional treatment, the effect of TeX processing is a DVI file containing *two pages*. The first one, generated by `\shipout`, will comprise the content of the box *without* the necessary header information. The next page will be generated by the `\end` instruction and it will contain the main vertical list (MVL) with the relevant header information.

There is no unique answer to this problem. It depends on our plans with respect to the header list. One of possible solutions is collecting all `\special` instructions from the main vertical list and adding them to all boxes (or only to the first box) by appropriately redefining the `\shipout` instructions.

Let's assume that the `\special` instructions defined in the `verbatimtex ... etex` block are put into the main vertical list without embedding them in boxes. Under this assumption, it is easy to distinguish a *header*-`\special` from the a *box*-`\special`. The first one is generated in external vertical mode, the latter one in internal vertical mode or horizontal mode (restricted or paragraph). To tell the difference it suffices to use a pair of conditionals `\ifinner` and `\ifvmode`. The macro `\mpspecial` defined in `cmdfont.tex` catches every `\special` that plans to visit the MVL and puts it into a special box, `\MVLspecialbox`. Note that specials are separated by a thin space (1sp), otherwise they would be glued together in the case of an `\unhbox` operation.

If our macros generated `\special` whatsits that are caught by `\mpspecial`, the `\pagetotal` register would be equal to 0pt. If this is not the case, it usually means that something went to the MVL, which is probably wrong. This case cannot be dealt with in a general way but if the total length of the MVL is still less than the page height we can try to pick it up and save into a box:

```
\par
\newbox \MVLbox
\begingroup
   \ifdim \pagetotal>0pt
      \errhelp{I'll save the MVL into MVLbox}
      \errmessage{MVL is not empty}
      \output={\global\setbox
         \MVLbox=\box255}
      \vsize=\pagetotal
      \eject
  \fi
\endgroup
```

It is up to the programmer what the `\MVLbox` is used for, once it is saved.

Recapitulation

Augmenting TEX, METAPOST, METAFONT, and some other programs with a specially treated font looks promising, especially with METAPOST.

Full evaluation of the new possibilities offered by the special font technique requires more experience than we as yet have. The technical details of the `cmdfont` structure need to be worked out. It is not obvious which parameters such a font should have. For instance, which size the space should have: 0pt, 1sp, $\frac{1}{3}$em (a typical value for a text font), or even 1em. We chose the latter because then it is easier to learn, from inside the TEX program, which is the value of the `designsize` parameter. It is also not obvious whether there should be only one `cmdfont` or a whole family of such fonts; and if so, which rules should be applied to avoid a mess.

A mess seems to be a critical threat to the effective application of the techniques described herein. Early standardization, including the font name, the details of the font design and the structure of texts typeset with it, is a *sine qua non* condition of success. We count on the TEX community—without their help it is unlikely that we manage to keep the mess away from this emerging technology which is still in its infancy.