Should TEX be Extended?

Michael Vulis
MicroPress Inc, 68-30 Harrow Street, Forest Hills, New York, 11375 USA
718-575-1816; FAX: 718-575-8038
Bitnet: cscmlv@ccnyvme

Abstract

This article examines three problems discussed in recent issues of TUGboat: Graphics inclusion, Font rotation, and Font selection scheme. The author compares the traditional solutions to the problems (pure TEX) to the solutions that can be obtained by slight extensions to either the TEX language primitives or the driver programs. For each problem, the article shows what can and cannot be achieved with puristic (Clean) TEX solutions; it will also describe how the limitations can be overcome with (Dirty) TEX language extensions and document the extensions.

On TEX

Since its inception eleven years ago, TEX has remained essentially unchanged. Meanwhile, the world of personal computing has advanced dramatically.

Circa 1980, a personal computer with 64k RAM was still considered advanced. Laser printers did not exist. WordStar and DisplayWrite were leaders in word processing. TEX was a revolution.

Circa 1985, PostScript was around, but prohibitively expensive. Proportional fonts were still a novelty. Desktop publishing was yet non-existent. Graphics was non-integratable. And TEX shined.

Circa 1990, leading word-processors (i.e., WordPerfect) format text almost as well as TEX, and perhaps easier. They handle graphics and tables much better than TEX, they generate indices and they spell check. They do not handle equations as well as TEX; however, they are not far off.

Circa 1995, TEX could become a historical curiosity.

On Extensions

Software systems that remain unchanged are destined for oblivion. TEX has lasted this long primarily because of its fresh start: immense superiority of TEX over other typesetting systems. This superiority is over, or almost over. To survive, TEX needs to evolve.

There are two ways the evolution of TEX can proceed: either one person, possibly even the Grand Wizard himself, can undertake serious and continuing rewriting of the system, or this rewriting will be done in possibly incompatible ways by several implementors. Since the Grand Wizard has declared his unwillingness to make any changes in the design, the second possibility appears likely. The goal of the TEX community should be to ensure that this rewriting does not get out of hand—to define the process of directing, implementing, documenting and *sharing* the extensions.

Historically, language compatibility has been assured by language standards. The existence of Standard (ANSI) Pascal, in particular, made TEX itself possible. A starting point, therefore, can be defining Standard TEX (TEX3.14159). TEX3.14159 will be identical to the TEX appearing in *The TEXbook*, with the following *change*: it will implement integer register `compatibility`. A 'TEX' can be deemed to be a 'TEX', if any source file that either starts with

```
\compatibility=0
  \let\compatibility\undefined
```

or does not include any of the new keywords should be handled identically by this TEX and TEX3.14159. Notice that this definition both supersedes the TRIP compatibility test and ensures that TEX documents can stay compatible between different systems.

On This Paper

With this definition in mind we will proceed with the study of a few changes to TEX that implement some of the desirable extensions. While the size of this paper will prevent us from presenting complete changes to the TEX code, these are available from the author (requestware). The extensions described

in this paper were implemented and tested under VTEX system (see *TUGboat*, August 1990). The four extensions discussed here include:

- ••Font rotation
- • Incorporation of graphics
- ••Automatic indices
- ••Font selection and/or substitution

Case Study I: Font Rotation in TEX

Of the three problems discussed in this article, font rotation probably received the least attention. The reason for it may be that before June 1990, no one has realized it was possible and afterwards no one thought it was practical. In June 1990, Alan Hoenig opened the chapter on Font Rotation with his beautiful examples (see *TUGboat*, 1990 Conference Proceedings) and closed the chapter with a scary explanation of how they were made.

Hoenig's approach consists of generating a series of TEX fonts via METAFONT, one font per required angle of rotation. For instance, to typeset a 24-character line of text around a circle, one would need to generate 24 variants of the original font. Similarly, a 100-character example requires 100 pre-generated fonts, 101-character example requires 101 different fonts ($gcd(100, 101) = 1$), while a 300-character can use the fonts generated for the 100-character example, but cannot be printed in most TEX versions (`font_max`$\leq$255). Hoenig's use of METAFONT was forced by two distinct reasons: drivers' inability to rotate fonts and, more to the point, TEX's inability to position characters when typesetting not on a horizontal line. To correctly update the reference point, TEX needs to know the sine and cosine of the typesetting angle; Hoenig made METAFONT compute them and pass them to TEX as extra `\fontdimen` parameters.

Hoenig's examples remain in the realm of curios, since it would not be practical to generate many fonts each time rotation is required. Even when drivers support font rotation (VTEX drivers do and PostScript drivers can), the problem remains as to how to compute sine's and cosine's. While it can be proven that macros for computing trigonometric functions can be written in TEX, a somewhat easier (and much faster) way is to simply transplant the relevant code (the `n_sin_cos` procedure) from METAFONT into TEX. In VTEX this is done by implementing a new `\sincos` primitive command and the `\sine` and `\cosine` dimen registers. Entering `\sincos1pt` fills `\sine` with $\sin(1°)$ and `\cosine` with $\cos(1°)$ (notice that one degree is one point). These values can be now used in typesetting computations. In addition, `\special{R###,###}` is used to tell the drivers about the desired rotation of the font. To avoid re-computing sines/cosines in drivers, we pass their values instead of the angle. Finally, we will need to somewhat modify Hoenig's macros:

```
{\catcode`p=12 \catcode`t=12
  \gdef\\#1pt{#1}}%
\let\getf=\\
\newdimen\x \newdimen\cos
\newdimen\y \newdimen\sin
\def\initialize{%
  \global\x=0pt\global\y=0pt}

\def\dolist{\afterassignment
  \dodolist\let\next= }
\def\dodolist{\ifx\next\endlist
  \let\next\relax
  \else \\\let\next\dolist\fi
  \next}
\def\endlist{\endlist}
\def\\{\expandafter\if\space\next
  \addspace\else\point\next\fi}
\newbox\spacebox
\setbox\spacebox=\hbox{\ }
\def\addspace{\setbox0=%
  \copy\spacebox\newcoords}
\def\point#1{%
  \setbox0=\hbox{#1}% for \newcoords
  \setbox2=\hbox{#1}% for typesetting
   \wd2=0pt \ht2=0pt \dp2=0pt
   \rlap{\kern\x \raise\y \box2}%
   \newcoords}
\def\newcoords{%
   \global\advance\x by  \cos
   \global\advance\y by -\sin}
\def\angletype#1{\initialize
   \leavevmode\setbox1=
     \hbox{\dolist#1\endlist}\box1}
```

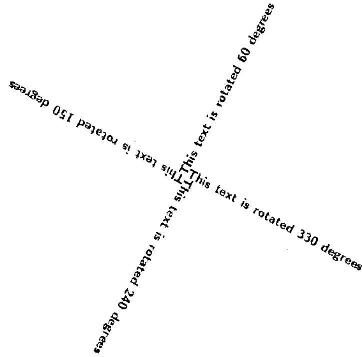
Now, we define

```
\def\tryrotation#1{%
 \setrotation #1pt%
 \def\sin{%
   \expandafter\getf\the\sine\wd0}%
 \def\cos{%
   \expandafter\getf\the\cosine\wd0}%
 \special{R\the\cosine,\the\sine}%
 \angletype{%
   This text is rotated #1 degrees}%
 \special{R0,0}}  % Turn off rotation.
```

and type

```
\vskip-1cm  \hskip8cm
\font\anglefont=mvssbx10 \anglefont
\tryrotation{60}%  Remove spaces to
\tryrotation{150}% keep the reference
\tryrotation{240}% point the same for
\tryrotation{330}% all four lines
```

to obtain



Other examples shown in Hoenig's article can be handled similarly.

Internals. The changes needed in the TEX program are as follows: define new dimension parameters `\sine` and `\cosine` (new codes are `sine_code` and `cosine_code`) and new `extension` primitive `\sincos` (using `compute_sincos` code) and accordingly modify `init_prim` and `print_cmd_chr`. Procedure `do_extension` receives new case:

```
compute_sincos:
  begin
  scan_normal_dimen; {angle*1000}
  n_sin_cos(cur_val*16);
  n_sin:=n_sin div 4096;
  n_cos:=n_cos div 4096;
  eq_word_define
    (dimen_base+  sine_code,-n_sin);
  eq_word_define
    (dimen_base+cosine_code, n_cos);
  end;
```

where `n_sin` and `n_cos` are temporary integers (in the METAFONT source, these were macros). Finally, transplant `n_sin_cos` as well as the procedures it needs (`pyth_add`, `make_fraction`, and `take_fraction`) from the METAFONT into the TEX source. This modification adds about 2K to the TEX program.

Case Study II: Bitmap Graphics Inclusion

The problem. A casual study shows that about 10% of articles published in TUGboat deal with graphics inclusion problems. This should not be unexpected since TEX's design completely ignores the existence of graphics. Graphics inclusion is normally done in one of two ways: either TEX allocates space for a graphics box, sets the reference point and passes the name of the graphics file via a `\special`, or graphics are converted into `.tfm/.pk` pairs and TEX treats them as characters. The advantage here is that off-the-shelf drivers can be made to print graphics; the disadvantage is the extra conversion pass and, frequently, the need to maintain two copies of the graphics file: in the initial and in the `pk` format. Further problems arise because of the TEX's limit on the number of fonts. Finally, the `.tfm/.pk` approach is not applicable to vector graphics formats. The `\special` approach requires a way to measure the dimensions of the graphics images; it also assumes that the drivers can read (and scale) graphics in several graphics formats (PCX and TIF to start with).

A possible interface for TEX follows:

```
\newdimen\graphX \newdimen\graphY
\newbox\gbox    % graphics box.
\def\scalegraph#1#2{%
\graphX=1in \divide\graphX by #1
  \multiply\graphX by \graphx
\graphY=1in \divide\graphY by #2
  \multiply\graphY by \graphy}}

\def\makepicbox#1#2#3{%
\sizegraph #3
\scalegraph{#1}{#2}
\setbox\gbox
  \hbox{\special{G,#1,#2,#3}}
\ht\gbox=\graphY
\wd\gbox=\graphX}

%Example:
%  \makepicbox{300}{300}{test.pic}
```

where `\graphx` and `\graphy` hold the pixel dimensions, set by `\sizegraph`; `\special{G...}` communicates the name of the graphics file to the drivers. The parameters to `\makepicbox` are the "natural" x- and y-resolutions of the picture; if they match the resolutions of the device driver, no scaling is needed.

From the TEX's point of view, the only interesting question is the implementation of `\sizegraph`. There are several ways:

(1) Hardwire the dimensions inside the TEX source; i.e.,

```
\def\scalegraph#1{\graphx640\graphy350}
```

(2) Read the dimensions from an \input file. If the graphics are stored in graph.pic, we assume that there is a header file graph.tex containing \graphx640\graphy350. \scalegraph, therefore, will change the file extension to .tex and \input the file. The problem with this solution is the need for the user to create and maintain the header files. As a minimum, one would require an auxiliary utility for determining the dimensions of graphics (call it SIZEGRAPH) and a MAKE program for ensuring that all header files are up-to-date.
(3) Implement \sizegraph as an extension primitive; make \graphx and \graphy dimen registers. This is the original approach used by VTeX. On the positive side, it eliminates the need for header files; on the negative, it burdens the TeX program with the need to know different graphics formats. Another hidden advantage over (2) is that TeX accumulates names of the \input files in its string pool; thus in (2) the string pool is likely to overflow on documents that include hundreds of pictures.
(4) A combination of (2) and (3). Keep a standalone SIZEGRAPH program and make TeX invoke it whenever it needs to get the dimensions of a graphics image. This appears to be the overall best solution, since SIZEGRAPH can now be independently maintained and the extension to TeX is both very small and very general.

The \exec Extension

VTeX extends TeX by adding the \exec primitive. \exec is implemented as a message command with code 2 (code 1 is \message and code 2 is \errmessage). \exec takes two arguments: the external program name and the argument string. Whenever VTeX encounters \exec, it stops TeXing and invokes the external program; it resumes the execution once it retains the control. The return code of the external program is reported in the \errno integer register. \exec allows the following implementation of \sizegraph:

```
\def\sizegraph#1{%
  \exec{sizegraph.exe}{#1 > temp.tex}%
  \if\errno0\input temp.tex\else ??? \fi
  }
```

While \exec provides possibly the best way for passing the graphics dimensions to TeX, it can also be used, for instance, to implement \sincos outside of TeX. Font substitution extensions described below can also be done by \exec'uting lookups into auxiliary tables. In fact, the \exec command is the ultimate extension: most other extensions discussed in this paper can be implemented through \exec; at the same time \exec does not seriously infringe on TeX syntax. As will be seen below, \exec can even be implemented without any modifications to TeX whatsoever.

The discussion will not be complete without mentioning the \command variant of \exec. Under MS-DOS, \command passes the command string to the command processor, rather than executing the program directly. Thus, \command can be used to execute internal commands.

```
\def\command#1{\exec
  {command.com}{/C #1}}
```

Case Study III: Automatic Index Generation

Another logical application of \exec would be an automated index for TeX. The index macros defined in the Appendix E of *The TeXbook* and actually used in formatting the *Computers & Typesetting* series provide excellent tools for generating indices. Unfortunately, these tools cannot be fully used from inside TeX since TeX lacks sorting abilities. Adding sort to TeX is an extension that the author would hardly advocate; VTeX's index is constructed by running an auxiliary IDXSRT program via \exec and than merging the results into the document (IDXSRT is capable of sorting and formatting indices in many different ways; in particular, it can remove multiple references to the same item that appears on one page.). The index is constructed by first using the \icopy and \iput macros, where \iput writes the argument into the index file, together with page and/or section number; \icopy is simply

```
\def\icopy#1{#1\iput{#1}}
```

When it is time to insert the index, we use \mergeindex:

```
\def\mergeindex{%
 \immediate\closeout\@indexfile%
 \command{idxsrt \indexparams\
   \@indexname eraseme.tex}%
 \input eraseme.tex
 \command{erase eraseme.tex}}
```

where \indexparams define the switches to be passed to IDXSRT.

Case Study IV: Font Substitution

In preparing a document, one often needs to change the size (or the attributes) of the font, *regardless of the font used*: it may be desirable to typeset footnotes at eight points and titles at fourteen, regardless of what font changes may appear in the document. For instance, in preparation of this article, the author was hoping to enter

```
|\head *The {\tt\char92exec} Extension*|
```

However, the `\head` macro scaled the roman font to 12 points and left teletype at 10 points (see previous page). The problem is unresolved in PLAIN TEX; LATEX 2.09 solves it by providing 800-line long table of font substitutions (`LFONTS.TEX`) plus repeated definitions of `\large`, `\huge`, etc., all over the style files. LATEX's solution is only partial: it does not support point sizes not explicitely listed in `LFONTS.TEX`; neither do LATEX's tables support non-cm fonts.

Most TEX users would find it greatly desirable to have compact and portable definitions of `\large`, `\small`, etc., that will support all TEX fonts. Since the need to support all possible fonts *precludes* usage of LATEX-style tables, the effect will be achieved by extending TEX. We add new integer register `\fontscale`. All the `setfont` commands are processed *relatively* to `\fontscale`. For example, if `\fontscale` is set to 1200, `\tt` will invoke teletype at 12, not at 10 points. We can now define

```
\def\huge{\fontscale=2400\setfonts}
\def\large{\fontscale=1200\setfonts}
\def\small{\fontscale=800\setfonts}
\def\tiny{\fontscale=514\setfonts}
\def\outl{\outline=1\setfonts}
\def\fillp#1{\fillpattern=#1\setfonts}
\def\setfonts{\the\font\setmathfonts}
\def\setmathfonts{%
 \textfont0\textfont0
 \textfont1\textfont1
 ...    ...    ...    ...
 \scriptscriptfont15\scriptscriptfont15}
```

and so on. Notice that after setting `\fontscale` we need to reissue the last font command (`\the\font`) to ensure that the current font changes.

Remark: The drawback of this definition of `\setfonts` above is the loading of math fonts caused by each font change switch regardless of whether math fonts will be needed. An alternative is to declare

```
\newif\ifnewmath
\def\setfonts{\the\font\newmathtrue}
\everymath{\ifnewmath\setmathfonts
   \newmathfalse\fi}
```

which will eliminate unnecessary font loads but may or may not conflict with other usage of `\everymath`.

Internals. `\fontscale` (and its companion `\bold`, `\smallcaps`, `\shadow`, `\outline`, `\fillpattern`, `\slant`, and `\aspect`) are simply additional integer parameters. As mentioned above, these are added by modifying the `init_prim` and `print_cmd_chr` routines. The standard values (set by IniTEX) are 1000 for `\fontscale` and `\aspect` and 0 for others.

The tricky part is the modification of the `prefixed_command` routine that handles font assignments. We start by replacing the standard

```
set_font:
  define(cur_font_loc,data,cur_chr);
```

with

```
set_font:
  define(cur_font_loc,data,
    fsubst(cur_chr));
```

The `fsubst` procedure returns with unmodified `cur_chr` if one of three events holds:

1) the program is run in TEX-compatibility mode, where fonts cannot be substituted;
2) its argument is the `nullfont` (`cur_chr=0`); or
3) all eight relevant integer registers (`\fontscale` through `\aspect`) hold default values.

If none of the above is true, `fsubst` retrieves the parameters for the font-in-question, multiplies the magnifications and the aspect ratios, adds the slants, and applies the exclusive-or to the remaining parameters. It next verifies that the font with required parameters has not yet been loaded and calls `read_font_info` to create it. Finally, it returns the font number obtained from `read_font_info`.

A similar change in the `def_family` subcase of the `prefixed_command` routine makes the `\textfont`, `\scriptfont`, and `\scriptscriptfont` relative.

Invisible fonts/color separation. An additional benefit is the ability to implement color separation via invisible fonts. Assuming that the `\fillpattern0` is 100that the `\fillpattern1` is 0

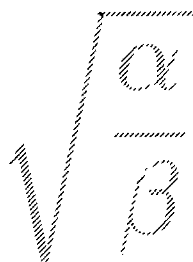
```
\def\print{\fillp{0}}
\def\dontprint{\fillpattern{1}}
\def\redcopy{
  \def\red{\print}
  \def\green{\dontprint}
  \def\blue{\dontprint}}
\def\bluecopy{
  \def\red{\dontprint}
```

```
\def\green{\dontprint}
\def\blue{\tprint}}
```

to allow selective printing of color planes.

Math rules. This pattern and color selection scheme needs a modification to be useful in math mode, where symbols are often built from both characters and *rules*. As given above, the `\fillp` command affects only the character part, creating misfits like

instead of



Currently, VTEX solves the problem by defining

```
\def\fillp#1{\fillpattern=#1\setfonts
  \special{F#1}}
```

where `\special{F#1}` instructs the device drivers to start shading rules.

Yet another difficulty is the possibility of shading that spans from one page to another. Unless the `\special` is re-issued on each page, a device driver would not see it if it processes the second page before the first. Solutions with different degrees of generality are possible.

A Special Note to a TEX Purist

Most of the extensions described in this paper can be used *without any changes* to TEX program. For instance, to use `\exec`, without implementing it we will write a loader program that traps screen and keyboard I/O and loads TEX, waiting for infamous

```
! Undefined control sequence.
<*> \exec
        {wipefile}{*.log}
?
```

(make sure that `\exec` and its arguments are on a line by itself, so they will be echoed on the next line.) The loader now swaps TEX out of memory, performs the `\exec`, swaps TEX in, and inserts `d8` into the TEX's mouth to delete now-unneeded tokens. While the author found this solution lacking in performance, it has been tested and worked with PC implementations of TEX.

A Late Note

After this paper has been presented at the TUG conference, a couple of participants noticed yet another usage for the `\exec`: as a security-breaching vehicle. Indeed, it is possible to write a TEX program to write, for example, a C program, and then `\exec` to compile, link and run it.

Conclusions

The author hopes that this paper will be helpful in encouraging further development of TEX.