	´0	´1	´2	´3	´4	´5	´6	´7	
´20x	Ç	ü	é	â	ä	à	å	ç	"8x
´21x	ê	ë	è	ï	î	ì	Ä	Å	
´22x	É	free	free	ô	ö	ò	û	ù	"9x
´23x	ÿ	Ö	Ü	free	ť	free	„	free	
´24x	á	í	ó	ú	ñ	Ñ	free	free	"Ax
´25x	free	free	“	ź	free	free	«	»	
´26x	ą	Ą	ă	Ă	ć	Á	Â	À	"Bx
´27x	Ş	Ć	č	Č	đ	Ż	ż	Đ	
´30x	đ	ę	Ę	ě	Ě	ğ	ã	Ã	"Cx
´31x	Ğ	ı	Ǐ	İ	Ĭ	Ľ	Í	Ĺ	
´32x	ð	Ð	Ê	Ë	È	Ň	Í	Î	"Dx
´33x	Ï	ł	Ł	ń	Ń	Ţ	Ì	ň	
´34x	Ó	free	Ô	Ò	ő	Ő	Š	Þ	"Ex
´35x	þ	Ú	Û	Ù	ý	Ý	ţ	ő	
´36x	Ő	ś	Ś	ş	š	Ť	ů	Ů	"Fx
´37x	Ű	Ÿ	Ź	ű	Ř	ř	ž	Ž	
	"8	"9	"A	"B	"C	"D	"E	"F	

The layout of the proposed CM font extensions

Janusz S. Bień

Fonts

Circular Reasoning: Typesetting on a Circle, and Related Issues

Alan Hoenig

Owing to the generality of both TEX and METAFONT, it's easy to typeset in and on circles. Here's how.

The METAFONT Part

TEX can't actually turn characters on their side; we ask METAFONT to create special fonts where each character in the font is rotated around its reference point (the lower left corner of the bounding box of any character). Then TEX properly positions characters from the rotated fonts to achieve the illusion of circular typesetting. We need one rotated font for each position on the circle.

What does it mean to typeset characters around the circumference of a circle? I imagined a regular

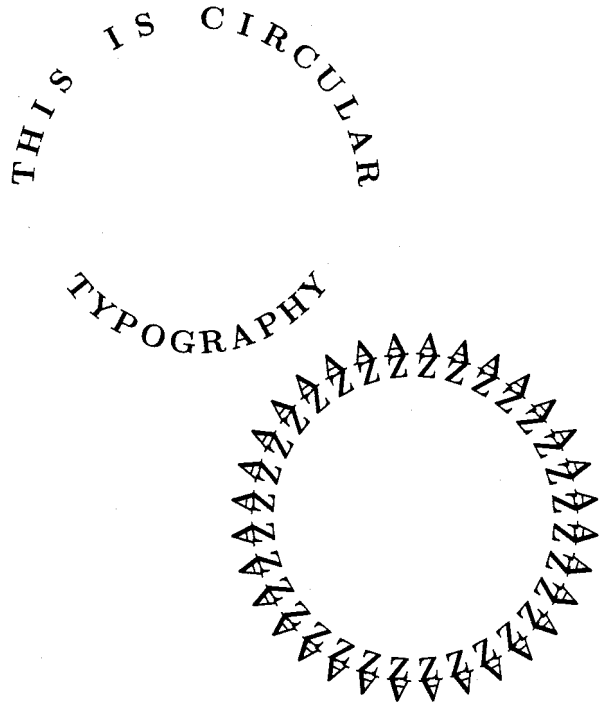

Figure 1. What this article is about.

polygon inscribed in the circle. The vertices of the polygon touch the circle from the inside, and the faces of the polygon form bases on which each character sits. Since each base is the same length as any other, I abandoned the concept of variable width typesetting on the circle; this accounts for the visually unsettling appearance of some circular typesetting. Later we will center each character on its base.

Let the bases be numbered from 0 to $n-1$; there are a total of n sides to this polygon. (It's more convenient to label the faces starting with 0 rather than 1.) Figure 2 shows a portion of such a circle with the first few faces. Notice that the zero-th face is at the "nine o'clock" position on the circle; that's because we read from left to right.

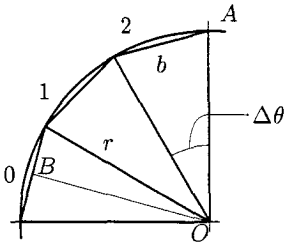

Figure 2. The inscribed n-gon on which we imagine placing rotated letters. The point B bisects its face.

For this article, I generated a sequence of rotated `cmbx12` fonts, and if we let $b = 12\,\text{pt}$, and imagine there to be room for 32 characters on the circumference of the circle, then the circle's radius must be 61.21 pt.

This follows from Figure 2 since

$$b/2 = r\sin(\Delta\theta/2)\,.$$

If n is the number of faces in the inscribed polygon, then $\Delta\theta = 2\pi/n$ or $\Delta\theta/2 = \pi/n$. Given $n = 32$ (then $\Delta\theta = 11.25°$) and $b = 12\,\text{pt}$, we must have $r \approx 61.21\,\text{pt}$.

Recall the way METAFONT files are organized. Parameter files (such as `cmbx12.mf`) call driver files (such as `roman.mf`), which contain further details about the organization of the particular font. Finally, this driver calls several program files containing the instructions for generating the actual characters in the font. We will need to make changes to the parameter and driver files; the program files remain untouched.

I took the file `cmbx12.mf` and made 32 copies of it, named `cmbx1200` through `cmbx1231`. The idea is that file `cmbx12`*nn*`.mf` generates the font whose letters are rotated to stand on face *nn* of our inscribed polygon. Make a copy of `roman.mf`, and call it `roroman.mf` (a rotated Roman font driver).

The changes to these files are essentially those which control the rotation of the font. The proper positioning of these characters involves knowledge of the trigonometric functions (sines and cosines) of certain angles. METAFONT does trig calculations very well, whereas TEX does them not at all. Therefore, we also need to pass the necessary trigonometric information to TEX for its use. We do this using the `fontdimen` mechanism.

Any font has several global characteristics that are helpful in typesetting. In a non-math font, these include things such as the width of a quad, the amount of stretchability of an interword space, and so on. These necessary quantities typically occupy positions `fontdimen1` through `fontdimen7`, but it's possible to create as many fontdimen parameters as needed. Note, for example, that if METAFONT stores a value of 1 (say) in `fontdimen10`, then TEX will read `\fontdimen10` for that font as 1 pt. TEX appends units of points to METAFONT's numerical fontdimen values.

Changes to Files CMBX12*nn*.MF

The parameter files need few changes. At the beginning of each file, modify the comments to remind yourself of the changes you will have made. I also adjusted the value of the parameter *ligs* = 0 to suppress ligatures. The last line of the file should be `generate roman;` change that to read

```
generate roroman.
```

The remaining changes are new lines which immediately precede this line, and they should look like this:

```
numeric wedge_angle;
 wedge_angle=360/32;
numeric face; face=0;
numeric rotation_angle;
 rotation_angle=
 90-(face+.5)*wedge_angle;
fontdimen9:  face, rotation_angle;
fontdimen11:
 sind wedge_angle, cosd wedge_angle;
fontdimen13: % for r=61.21pt
 sind(rotation_angle),
 cosd(rotation_angle);
fontdimen15: % for r=30.61pt
 sind(90-2(face+.5)*wedge_angle),
 cosd(90-2(face+.5)*wedge_angle);
fontdimen17: % for r=15.30pt
 sind(90-4(face+.5)*wedge_angle),
 cosd(90-4(face+.5)*wedge_angle);
```

This puts various parameters in `fontdimens` nine through seventeen. The rotation angle is the angle by which we need to rotate a letter from the vertical so it will sit on its proper face on the underlying n-gon. The rotation is done in a counter-clockwise direction, as per the usual METAFONT convention. In figure 2, angle $\overline{AOB}$ is the rotation angle for the letter that will sit on face 1. Notice that line OB bisects the wedge angle and is perpendicular to the face, which it bisects.

These lines should be the same in all of the rotated font parameter files, except for the line defining the value of `face`. In file `cmbx12`*nn*, the appropriate definition should be `face=`*nn*.

Changes to `roroman.mf`

METAFONT can rotate the elements it draws as a matter of course, so we need only the following few alterations to `roroman.mf`.

```
currenttransform:=currenttransform
 rotated rotation_angle;
def t_=transformed
 currenttransform enddef;
```

These statements should appear immediately following the line

```
mode_setup; font_setup;
```

and in any case before the sequence of `input` statements that follows.

METAFONT's `currenttransform` applies a transform to all the pictures it generates. We simply define this transform to include a rotation by the current value of the rotation angle, and METAFONT does the rest.

Thirty-Two New Fonts

Now, generate 32 new fonts. The METAFONT command line you need is

```
mf \&cm \mode=corona; input cmbx1200
```

and so on for the remaining 31 fonts. Minor variations will be necessary depending on your particular system. For example, you will need to select the proper `mode` name. In PCMETAFONT, for example, you conclude the line with the switches `/a=99/t`. Don't forget to change `input cmbx1200` to `input cmbx1201`, and so on. After creating each METAFONT font file, you need to transform the generic font file to a `pk` file via the utility `gftopk`; typically the command line to do that looks something like

```
gftopk cmbx1200.300 cmbx1200.pk
```

Finally, move the `tfm` file to wherever all your other `tfm` files are (probably in a directory named something like `\tex\textfms`) and move the `pk` files to their proper directory, something like `tex\pixel\dpi300` for laser printer fonts; change the '300' to the resolution of your printer. (If your pixel files are organized according to the older convention involving numbers like 1500 and so on, the determination of where to place these fonts is less straightforward. In general, though, these font files should reside in the same region of your hard disk as do the fonts you use for normal 10 pt, `\magstep0` typesetting.)

I confess I only generated the uppercase letters to these rotated fonts to save my time and disk space. If you elect to follow suit, you'll have some minor additional changes to make to `roroman`—namely, comment out all but the first `input` statement in that file. You'll probably want to create batch files to generate your fonts, convert them to `pk` form, and move them to the proper directories.

A TEX Digression

As a warmup exercise we can do something simpler than circular typesetting. We will first typeset on an angle. To typeset up a 45-degree incline, we need a special font which I named `zcmr10`. I deviated from my naming scheme because no face is inclined at the proper angle when there are 32 faces in the polygon. In `zcmr10.mf`, let the rotation angle be 45 (degrees). Most of the TEX macros that are responsible for placing the letters properly appear somewhere in *The TEXbook*; as is so often the case, doing something interesting with TEX is a matter of the artful extraction of the relevant bits and pieces from *The TEXbook*.

The macros depend on a `\getfactor` macro. It takes a single argument, namely a particular fontdimen for a certain font, and returns the value of that fontdimen stripped of the units of points. This macro is largely adapted from an example in Appendix D (page 375). Watch closely.

```
{\catcode`p=12 \catcode`t=12
 \gdef\\#1pt{#1}}
\let\getfactor=\\
```

Thus, if `\the\fontdimen1\tenit` is '0.25pt', then

```
\getfactor\the\fontdimen1\tenit
```

will yield 0.25 in some context where 0.25 makes sense.

We have to sidestep TEX's typesetting mechanism, since we are not setting characters on a common baseline, and we appropriate part of the solution to exercise 11.5, in which we learn how to seize individual tokens in a list. Here's the relevant code.

```
\def\dolist{\afterassignment
 \dodolist\let\next= }

\def\dodolist{\ifx\next\endlist
 \let\next\relax
 \else \\\let\next\dolist \fi
 \next}
\def\endlist{\endlist}

\def\\{% next char letter or space?
 \expandafter\if\space\next\addspace
 \else\point\next\fi}
```

Macro `\addspace` (see below) is responsible for leaving spaces in the angle copy. The macro `\point`, drawn from Appendix D, is used to position the current character. In order to use these macros, we need to initialize certain registers and fonts.

```
\newdimen\x \newdimen\y
\def\initialize{\global\x=0pt
                \global\y=0pt }
```

We will depend on `\newcoords` to compute the coordinates for the reference point of the next character. We use analytic geometry to determine

$$\Delta x = -\sin\theta\,\backslash\texttt{wd0}$$
$$\Delta y = \quad \cos\theta\,\backslash\texttt{wd0}$$

where θ is the angle of inclination of the type from the vertical (here $\theta = 45°$) and `\wd0` is the width of the current character or space which is in `\box0`. Then, $x \leftarrow x + \Delta x$ and $y \leftarrow y + \Delta y$.

```
\font\anglefont=zcmr10 % rotated font
\newdimen\DeltaX \newdimen\DeltaY
\def\newcoords{%
  \DeltaX=\expandafter\getfactor
  \the\fontdimen14\anglefont \wd0
  \DeltaY=\expandafter\getfactor
  \the\fontdimen13\anglefont  \wd0
  \global\advance\x by-\DeltaX
  \global\advance\y by\DeltaY }
```

`\getfactor` strips the 'pt' from fontdimens 13 and 14 and uses the resulting numbers — values of sine and cosine for an angle — as coefficients of the width of the box containing a space.

Here is the TEX code for `\addspace`, which determines how much space to leave between words.

```
\newbox\spacebox
 \setbox\spacebox=\hbox{\ }
\def\addspace{\setbox0=
 \copy\spacebox \newcoords}
```

The `\point` macro that I use is slightly different from the one Donald Knuth provides in Appendix D. Here is its code.

```
\def\point#1{%
  \setbox0=\hbox{\anglefont #1}%
   % used by \newcoords
  \setbox2=\hbox{\anglefont #1}%
   % used for typesetting
  \wd2=0pt \ht2=0pt \dp2=0pt
  \rlap{\kern\x \raise\y \box2}%
  \newcoords}
```

Finally, the `\angletype` macro puts all the pieces together.

```
\def\angletype#1{\initialize
  \leavevmode\setbox0=
    \hbox{\dolist#1\endlist}%
  \box0 }
```

The instruction `\angletype{Angle of Repose}` was sufficient to typeset the subject of Figure 3.

Figure 3. Typesetting at an angle.

Angle typesetting might be useful when you prepare advertising copy, and when you need to typeset column headings on tables with very narrow columns, as in Figure 4.

Typesetting on Circles

Once the angle-setting macros are in place, we need to alter details to accomplish typesetting on a circular path. On a circle, things change as we move along the circumference — we have to keep track of our position along the circumference, and at each new face we have to select the appropriate font.

The macros `\getfactor`, `\dolist`, `\dodolist`, and `\\` remain the same. (In `\\`, we rename

	Port	Claret	Burgundy	Champagne
1986	0	6	6	0
1985	0	7	7	7
1984	0	4	4	0
1983	7	6	7	6

Figure 4. A portion of a table with narrow columns. This is a portion of a table showing quality of recent vintages. The numbers give quality in a scale of 1 through 7; 0 means the wine is unrated.

\addspace to \newcoords.) The first new macro will determine the coordinates to the next vertex of the underlying polygon on which we place each type. We identify these coordinates as x_i and y_i. First we initialize the coordinates. The initial vertex (x_0, y_0) has coordinates $(-r, 0)$. \faceno is a numeric register containing the current face number (recall that we draw a correspondence between position along the circumference and a particular face of the inscribed 32-gon). Various radius-like quantities will later enable us to typeset around circles of varying radius.

```
\newcount\faceno
\newdimen\Bigradius \newdimen\Radius
\newdimen\radius \newdimen\r
\Bigradius=61.21pt
\Radius=.5\Bigradius
\radius=.5\Radius
\r=\Bigradius
\newdimen\x \newdimen\y
\def\initialize{\font\anglefont=
  cmbx1200 \global\faceno=0
  \x=-\r \y=0pt }
```

Given vertex (x_n, y_n), we can get the next vertex (travelling clockwise) via

$$x_{n+1} = x_n \cos\Delta\theta + y_n \sin\Delta\theta$$
$$y_{n+1} = -x_n \sin\Delta\theta + y_n \cos\Delta\theta$$

(see, e.g., David Salomon's article in *TUGboat* 10, no. 2, p. 213, July, 1989). We calculate these quantities using registers \dimen0, \dimen1, and \dimen2.

```
\def\nextpointt{%
  \dimen0=\expandafter\getfactor
    \the\fontdimen12\anglefont \x
  \dimen2=\expandafter\getfactor
    \the\fontdimen11\anglefont \y
  \advance\dimen0 by\dimen2
  \dimen1=\dimen0
  \dimen0=-\expandafter\getfactor
    \the\fontdimen11\anglefont \x
  \dimen2=\expandafter\getfactor
    \the\fontdimen12\anglefont \y
  \advance\dimen0 by\dimen2
  \global\x=\dimen1 \global\y=\dimen0}
\def\nextpoint{\nextpointt
   \preparefornextface}
\let\newcoords=\nextpoint

\newcount\lastface \lastface=31
\def\preparefornextface{%
  \global\advance\faceno by 1
  \ifnum\faceno>\lastface
    \global\faceno=0
   \message{There may be too many
    letters in your circular message!}%
  \else \ifnum\faceno<10
    \font\anglefont=cmbx120\the\faceno
  \else \font\anglefont=cmbx12\the\faceno
  \fi \fi}
```

Macro \preparefornextface changes fonts for the next face of the underlying polygon, and uses a numerical register \faceno for that purpose.

We won't use the coordinates (x_i, y_i) for typesetting, because that would put the reference point of the type at the vertex of our underlying, imaginary 32-gon. It is much better to center the type on its base. The centering macro \setonbase assumes that \box2 contains the current character and the corrected coordinates are (x'_i, y'_i)

If w is the width of the type and b is the length of the base, then the vector $\Delta\mathbf{r}$

$$\Delta\mathbf{r} = \frac{b-w}{2}(\cos\theta, \sin\theta)$$

provides the correction to $\mathbf{r} = (x_i, y_i)$ so that if we place the reference point of the type at the point $\mathbf{r}' = \mathbf{r} + \Delta\mathbf{r}$, then it will be centered on that base. θ is the rotation angle.

We can easily get $\Delta\mathbf{r}$ from $\mathbf{r}$ since the two vectors are perpendicular to each other. Given that $\mathbf{r} = r(-\sin\theta, \cos\theta)$, then either of $\pm(\cos\theta, \sin\theta)$ are perpendicular to it. Since $\Delta\mathbf{r}$ represents an offset in the clockwise direction, we choose the + sign.

```
\newdimen\xprime \newdimen\yprime
\def\setonbase{% curr char in \box2
  \xprime=\x \yprime=\y
  \baseoffset=.5\base
   \advance\baseoffset by-.5\wd2
  \dimen0=\expandafter\getfactor
   \the\fontdimen14\anglefont \baseoffset
  \dimen2=\expandafter\getfactor
```

```
 \the\fontdimen13\anglefont \baseoffset
\advance\xprime by\dimen0
\advance\yprime by\dimen2 }
```

Finally, we need a slightly altered \point macro, and \circumtype puts everything together.

```
\newdimen\base \base=12pt
\newdimen\baseoffset
\newtoks\currchar
\def\point#1{%
  \currchar=\expandafter{#1}%
  \setbox2=\hbox{\anglefont
  \the\currchar}%
\setonbase
\wd2=0pt \ht2=0pt \dp2=0pt
\rlap{%
  \kern\xprime \raise\yprime\box2}%
\newcoords}

\def\circumtype#1{%
  \initialize
  \setbox0=\hbox{\dolist#1\endlist}%
  \leavevmode \box0 }
```

Figure 5 shows the alphabet around a circle. If the irregular rhythm of the type due to placing variable width type at equal intervals bothers you, you might want to consider using a monospaced font like cmtt10 instead of the cmbx12 that I used.

Figure 5. Circular typesetting.

Smaller Circles

Because 32 is divisible by four, it is easy to typeset on circles that are one-half and one-quarter the radius of the original circle. Such cartouches would accommodate 16 and 8 characters around their circumferences. To do this right, we would need additional trigonometric values in cmbx12*nn*.MF so that an enhanced version of \setonbase can compute Δr properly. That's why we included information for fontdimens 15 through 18 in the METAFONT parameter files.

Actually, the changes to \setonbase are extensive and I have not done them at this time. If you decide you want to, here are some things to keep in mind. When we shrink the radius, we need to increase the wedge angle. Halving the radius requires doubling the wedge angle (provided the length of the base remains constant), and so on. At a half radius, for example, we skip every other vertex of the original 32-gon. In Figure 6, we set a letter on faces $\overline{AB}$ and $\overline{CD}$ (closer to the center of the circle, though) while skipping faces $\overline{BC}$ and $\overline{DE}$. However, we need fontdimen information from the skipped fonts to get information about the vectors $\mathbf{\Delta r}$. In Figure 6, OB is perpendicular to AC. We need to invoke and save information from that skipped font.

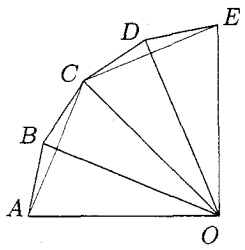

Figure 6. Typesetting when we change the radius.

However, it's easy to do "poor man's" typesetting around smaller circles if we adopt a "dummy" version of the \setonbase macro. Here's all we need do.

```
\r=\Radius
\def\newcoords{\nextpoint
  \nextpoint}
\def\setonbase{% dummy def'n
  \xprime=\x \yprime=\y}
```

To typeset around the smallest circle, simply set

```
\r=\radius
```

and

```
\def\newcoords{\nextpoint \nextpoint
 \nextpoint \nextpoint }
```

Because of the do-nothing version of \setonbase, the reference point of each letter coincides with the

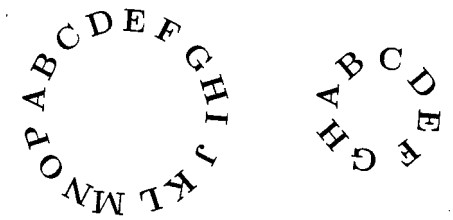

Figure 7. Typesetting around small circles.

vertices of the underlying 16-gon or octagon. Somehow, though, at smaller radii, this is less visually unsettling than we would expect (see Figure 7).

On the Inside of a Circle

Suppose we wanted to typeset around the *inside* of a circle. In light of the foregoing, one approach is to simply "METAFONT up" a new set of 32 fonts using a slightly different expression for the rotation angle, but it is possible to use the fonts we already have. For example, on face 1, we use font `cmbx1201` to determine type placement information, but we typeset the letter using the font that would normally appear diametrically opposite it (in this case, face 17). Given a face n, its opposite face $n_{\rm opp}$ must satisfy

$$|n - n_{\rm opp}| = N/2$$

where N is the total number of faces and both n and $n_{\rm opp}$ must be non-negative integers less than N. (Remember, $N = 32$ for our largest circle.)

When we use the font that belongs at the opposite face, we need to keep two points in mind. First of all, the reference point of the opposite font lies not at vertex n, but at vertex $N+1$, the next clockwise vertex (think about it). We can take this into account in our initialization macro. When using opposing fonts, the initial position of the type on face 0 is not at $(-r, 0)$ but at $(-r\cos\Delta\theta, r\sin\Delta\theta)$.

Because of the displacement of the reference point, the vector Δr that the `\setonbase` macro uses must point in the counterclockwise direction. These changes dictate the following new macros which contribute to the construction of macro `\circintype`.

```
\newcount\oppface
\def\setinbase{%
  \xprime=\x \yprime=\y
  \baseoffset=.5\base
   \advance\baseoffset by-.5\wd2
  \dimen0=\expandafter\getfactor
   \the\fontdimen14\anglefont\baseoffset
  \dimen2=\expandafter\getfactor
   \the\fontdimen13\anglefont\baseoffset
  \advance\xprime by-\dimen0
  \advance\yprime by-\dimen2
  \oppface=\faceno \advance\oppface by0
  \ifnum\faceno<16
   \advance\oppface by16
   \else \advance\oppface by-16
   \fi
  \ifnum \oppface<10
  \font\oppfont=cmbx120\the\oppface
   \else
  \font\oppfont=cmbx12\the\oppface
  \fi \setbox2=
    \hbox{\oppfont \the\currchar}}

\def\ininitialize{\font\anglefont=
   cmbx1200 \global\faceno=0
  \x=-\expandafter\getfactor
   \the\fontdimen12\anglefont \r
  \y= \expandafter\getfactor
   \the\fontdimen11\anglefont \r }

\def\circintype#1{\bgroup
  \let\setonbase=\setinbase
  \let\initialize=\ininitialize
  \initialize
  \setbox0=\hbox{\dolist#1\endlist}%
  \leavevmode\box0 \egroup}
```

Figure 8 shows what to expect from inscribed circular typesetting.

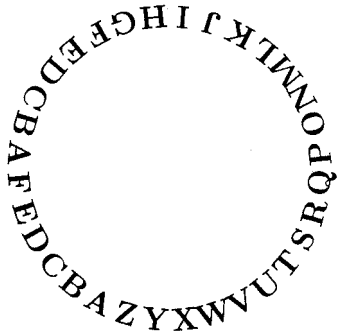

Figure 8. Typesetting inside a circle.

Incidentally, here are the commands I used to generate the top of Figure 1.

```
\circumtype{%
 THIS IS CIRCULAR~~~~~~~~~~~~~~~~}%
\circintype{%
 ~~~~~~~~~~~~~~~~~~~YHPARGOPYT~~~}%
```

I suffered plenty of setbacks *en route* to a working set of circular macros. Sometimes the results of faulty macros were interesting in their own right. Take a look at the best such mistake in Figure 9.

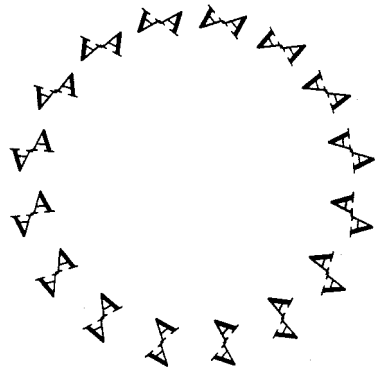

Figure 9. Mistake.

If you try this stuff yourself, note that circular typesetting may throw your previewer and device driver for a loop (apt?). You have been warned.

◇ Alan Hoenig
17 Bay Avenue
Huntington, NY 11743
(516) 385-0736

Graphics

On the Implementation of Graphics into TEX

Gerhard Berendt

1 Abstract

The problem of implementing more complex pictures than are provided by the LaTeX `picture` environment into a typical PC version of TEX is discussed. In the first part of the article (Sections 2 and 3) a solution is presented which circumvents the usual limitation of the restricted main memory of TEX and respects the moderate hash size of the PC versions of TEX. This solution remains, however, totally within the frame of TEX. In the second part (Sections 4 to 7) a solution to the problem is given which makes use of PostScript within the TEX environment.

2 Introduction

While TEX is a very powerful tool for producing mathematical and technical texts, it has its well-known deficiencies as far as the implementation of graphics is concerned. The problem is twofold:

- The hash size of about 3000 for a typical PC version of TEX limits the complexity of macro packages which implement graphics. It is, e.g., impossible to add the rather comfortable PICTEX macro[1] package to LaTeX because of an overflow of the hash size. In order not to surpass the given hash size, it is therefore necessary to use a more moderate graphics macro package, if the LaTeX environment is obligatory. Our solution to this problem will be presented in the next section.
- Another more subtle problem results from the fact that even a picture of only moderate complexity — if it is not produced by characters of special fonts (as is the philosophy in LaTeX) — might overflow the main memory of TEX. It is then impossible to compile a page which contains this picture. The only way out of this difficulty is to compile text and picture separately and either to combine the two `dvi` files afterwards or to print text and picture in two runs.

In the first part of this article, we present a compromise solution to both problems which:

- enables the user to produce texts plus included pictures of moderate complexity; and
- needs nothing but LaTeX running on a PC together with a small graphics macro package, a parameter file extraction program and (optionally) another utility program which automates the creation of the picture input.

Our solution relies neither on special output devices or files (e.g. laser printers or PostScript files) nor on drawing programs or special picture formats. Instead, the pictures are drawn within the LaTeX `picture` environment which is enriched by a few graphics macros from the extended `epic` style.

[1] M.J. Wichura, *TUGboat* 9, no. (2), p. 193, 1988