

T_EX beauties and oddities

A permanent call for T_EX pearls

What is wanted:

- ▷ short T_EX or METAPOST macro/macros (half A4 page or half a screen at most),
- ▷ the code should be generic; potentially understandable by plain-oriented users,
- ▷ results need not be useful or serious, but language-specific, tricky, preferably non-obvious,
- ▷ obscure oddities, weird T_EX behaviour, dirty and risky tricks and traps are also welcome,
- ▷ the code should be explainable in a couple of minutes.

The already collected pearls can be found at <http://www.gust.org.pl/pearls>. All pearl-divers and pearl-growers are kindly asked to send the pearl-candidates to pearls@gust.org.pl, where Paweł Jackowski, our pearl-collector, is waiting impatiently. The pearls market-place is active during the entire year, not just before the annual BachoT_EX Conference.

Note: The person submitting pearl proposals and/or participating in the BachoT_EX pearls session does not need to be the inventor. Well known hits are also welcome, unless already presented at one of our sessions.

Since some seasoned T_EX programmers felt indignant of calling ugly T_EX constructs “Pearls of T_EX programming”, we decided not to irritate them any longer. We hope they will accept “T_EX beauties and oddities” as the session title.

If you yourself have something that fits the bill, please consider. If you know somebody’s work that does, please let us know, we will contact the person. We await your contributions even if you are unable to attend the conference. In such a case you are free either to elect one of the participants to present your work or “leave the proof to the gentle reader” (cf. e.g. <http://www.aurora.edu/mathematics/bhaskara.htm>).

Needless to say that all contributions will be published in a separate section of the conference proceedings, possibly also reprinted in different T_EX bulletins.

Taco Hoekwater

Suffer from suffix

When a font is defined, TeX scans but ignores a filename extension. A side effect of that is that the code below works to select `texnansi-lmr10.tfm`

```
\font\test=texnansi-lmr10.WhoToldYouToUseASuffix?
\test oeps
```

And depending on whether your local Web2c version is compiled with or without multiple file extensions support, so may this:

```
\font\test=texnansi-lmr10.4pt.
\test oeps
```

Javier A. Múgica

\if plaiting

Suppose we have

```
\let\ifouter=\iftrue
\ifouter <true code> \else <>false code> \fi
```

and suppose that in true block there is another `\if` test that has to be built using `\csname..\endcsname`.

```
\expandafter\let\csname if@inner\endcsname\iftrue
```

Does the following work?

```
\ifouter
  \csname if@inner\endcsname
  <double true code>
  \csname fi\endcsname
\else
  <>false code>
\fi
```

Only if the inner test is true. Otherwise the condition will not be balanced. But we can plait `\if` blocks as in

```
\ifouter
  \iftrue
  \csname if@inner\endcsname
  <triple true code>
  % \else % does not spoil the game
  \fi
  \csname fi\endcsname
\else
  <>false code>
\fi
```

Javier A. Múgica

Access outer `\input` level

Imagine we need to grab the first param from some input `file.tex`. We are lucky, the following works fine:

```
\def\do#1{<do something with #1>}
\expandafter\do\input file.tex
```

And what if we would like to execute `\do` from the `file.tex` level, catching the parameter that lays just after the `\input` command in the outer level document? Well, since `\do` requires an argument, it cannot occur at the very end of the input `file.tex`. We may try this:

```
% outer document content:
\def\do#1{#1 indeed}
\input file.tex {crazy}?\end

% file.tex content:
Isn't that \expandafter\do\endinput
```

But then \TeX complains:

```
! File ended while scanning use of \do
```

So we don't expand `\endinput`, but the end of the file itself:

```
% outer document content the same as before,
% file.tex content:
Isn't that \expandafter\do
```

Isn't that crazy indeed?

And then it works! (The actual example is in my package `makepattern`, file `mkpatter.tex` in CTAN.)

Hans Hagen

Crash on recurse

Check this out:

```
%\newtoks \test % as it was originally surfaced
\def\test{\the\test}
\test
```

We might expect an endless loop here. But we got an unexpected \TeX crash instead.

Look at that:

```
\def\test{\show\test} % \def\test{\showthe\test}
\test
```

gives

```
> \test=macro:
->\show \test .
\test ->\show \test
```

Usual in a language a definition binding to a name happens afterwards, but not in \TeX ...

Paweł Jackowki

\badness badness

\badness parameter is supposed to tell something about box stretch or shrinkage, or more precisely, about how far the available stretchability or shrinkability has been used. \badness=0 means no stretch or shrink used, 100 means the entire flexibility consumed, more then 100 denotes underfull or overfull box (the later with \badness equal 1 000 000). So \badness certainly says something about how the content fits the box, but one may be surprised by fine artifacts of the actual algorithm. Lets take a macro that measures \vbox \badness, taking the actual and the available stretch as arguments.

```
\def\test#1#2{% <actual_stretch> <available_stretch>
  \setbox0\vbox to\dimexpr#1{\vskip0pt plus\dimexpr#2}%
  \immediate\write16{badness=\the\badness}}
```

```
\test{100pt-1sp}{100pt} % \badness=99
\test{100pt}{100pt}      % 100
\test{100pt+1sp}{100pt} % 100
\test{100.3366pt}{100pt} % 100
\test{100.3367pt}{100pt} % 101
```

Note that we never get \badness equal 100 if the actual stretch is even 1sp smaller then the available stretch. But T_EX seems to ignore the stretch abuse around 0.0033 and less.

OK so far. Now lets take such insignificantly underfull box of 299sp actual, and 298sp available stretch. We will scale those values with integer in factor.

```
\def\scale#1{\test{299sp*#1}{298sp*#1}}
\scale{1} % 100
\scale{2} % 100
\scale{3} % 100
...
\scale{24182} % 101
\scale{24184} % 101
...
\scale{24948} % 100
```

\badness varies although we keep the stretch ratio fixed!

All because heuristic \badness formula (`tex.web`, line 2320):

```
...
begin if t=0 then badness:=0
else if s<=0 then badness:=inf_bad
else
  begin if t<=7230584 then r:=(t*297) div s {$297^3=99.94\times2^{18}$}
  else if s>=1663497 then r:=t div (s div 297)
  ...
```

Karl Berry

Word double-hyphenation

How about this macro for an explicit hyphen that does not inhibit further hyphenation within the word? (It's not original with me; the $\text{T}_{\text{E}}\text{X}$ nique was first described by Knuth, I believe.)

```
\def\xhyphen{\leavevmode\nobreak-\hskipOpt }
```

Let's compare the following two paragraphs:

```
\hsize=2.5in
```

How about this macro for an explicit hyphen that does not inhibit further-hyphenation within the word?

How about this macro for an explicit hyphen that does not inhibit further\xhyphen hyphenation within the word?

How about this macro for an explicit hyphen that does not inhibit further-hyphenation within the word?

How about this macro for an explicit hyphen that does not inhibit further-hyphenation within the word?

All the magic is done by the `\hskipOpt`, which essentially starts a new word as far as $\text{T}_{\text{E}}\text{X}$ is concerned (in the middle of the hyphenated word), thus giving it a new chance at hyphenation. The `\leavevmode` merely avoids the penalty being added to the vertical list, in case the macro is used at the beginning of a paragraph.

David Kastrup

Is $2\text{in} = 1\text{in} + 1\text{in}$?

$\text{T}_{\text{E}}\text{X}$'s arithmetic forgets to round when dealing with units. For example, `1in` is defined to be exactly 72.27pt. We can check that by writing `\dimen0=100in \the\dimen0` and getting 7227.0pt. So far, so good. But when we write `\dimen0=1in \the\dimen0`, we get 72.26999pt. Oops. This becomes worse since a number called 72.27pt actually exists, as witnessed by `\dimen0=72.27pt \the\dimen0` which gives us 72.27pt.

So we have the unfortunate situation that `1in` (which should be exactly 72.27pt) gives a number different from the number actually called 72.27pt. This is because $\text{T}_{\text{E}}\text{X}$ truncates when working with the fractions representing exact units, but rounds when working with decimal fractions.

$\text{T}_{\text{E}}\text{X}$ calculates `1in` as

$$\left[1 \times \frac{7227}{100} \times 2^{16} \right] 2^{-16}$$

namely truncating the fractional calculation rather than rounding it.

Other artifacts of $\text{T}_{\text{E}}\text{X}$'s fractional representation of units mean that $2\text{in} \neq 1\text{in} + 1\text{in}$: Indeed

```
\dimen0=1in \advance\dimen0 1in \the\dimen0 \dimen0=2in =\the\dimen0
```

leaves us with $144.53998\text{pt} = 144.54\text{pt}$.

David Kastrup

Fixing `\vtop`

`\vtop` happens to lose the correct top of line information when it starts with a `whatsit`. We can reconstitute this in the following manner:

```
\def\fvtop#{\vtop\bgroup
  \setbox0\vbox\bgroup
  \aftergroup\fvboxii
  \let\next=}

\def\fvboxii{\setbox0\vtop{\break\unvbox0}%
  \dimen0=\maxdimen
  \ifdim\dp0>0pt \advance\dimen0-\dp0 \fi
  {\splittopskip\dimen0 \setbox0\vsplit0to0pt}%
  \advance\dimen0-\ht0%
  {\splittopskip-\maxdimen\setbox0\vsplit0to0pt}%
  \advance\dimen0\ht0%
  \hrule height\dimen0 depth-\dimen0
  \unvbox0
  \egroup}
```

The trick here is to use `\vsplit` which pads the followup box sufficiently to reach a given line height on the top line. We essentially tell it to make this line fit `\maxdimen` minus a safety margin and see how far it gets. Then we remove the padding with another split and start the box with a strut that establishes the corrected height.

```
\def\example{\hbox{First good line}%
\hbox{Second good line}%
\hbox{Third good line}}

\leavevmode
\vtop{\example}%
\vtop{\write-1{\example}%
\fvtop{\write-1{\example}}
```

```
First good line First good line First good line
Second good line First good line Second good line
Third good line Second good line Third good line
Third good line Third good line
```