BIJLAGE X

The structure of the TEX processor [1] [2]

Victor Eijkhout

Center for Supercomputing Research and Development
University of Illinois
305 Talbot Laboratory
104 South Wright Street
Urbana, Illinois 61801-2932, USA
eijkhout@csrd.uiuc.edu

Februari, 1991

The inner workings of TEX are explained by its author [1] in terms of an analogy with the digestive tract. Apart from the fact that this gives rise to a whole genre of jokes[3], the analogy becomes definitely strained when regurgitation takes place in the mouth, or when the eyes take part in the process.

In this article I will describe the TEX processor as a multi-layered engine that successively transforms characters into tokens, tokens into lists, and from these lists builds a typeset page.

1 Four TEX processors

The way TEX processes its input can be viewed as happening on four levels. One might say that the TEX processor is split into four separate units, each accepting the output of the previous stage, and delivering the input for the next stage. The input of the first stage is then the `tex` input file; the output of the last stage is a `dvi` file.

For many purposes it is most convenient and insightful to consider these four levels of processing as happening after one another, each one accepting the *completed* output of the previous level. In reality this is not true: TEX is not something like a four-pass compiler. All levels are simultaneously active, and there is interaction between them.

The four levels are

1. The input processor. This is the piece of TEX that accepts input lines from the file system of whatever computer TEX runs on, and turns them into tokens. These are mostly character tokens that comprise the typeset text, and control sequence tokens that are commands to be processed by the next two levels.
2. The expansion processor. A number of tokens generated in the first level – macros, conditionals, and a number of primitive TEX commands – is subject to expansion. Expansion is the process that replaces some (sequences of) tokens by another (possibly empty) sequence.
3. The execution processor. Control sequences that are not expandable are executable, and this execution takes place on the third level of the TEX processor.
   One part of the activity here concerns changes to TEX's internal state: assignments and macro definitions are typical activities in this category. The other thing going on on this level is the construction of horizontal, vertical, and mathematical lists.
4. The visual processor. In the final level of processing the visual part of TEX processing is performed. Here horizontal lists are broken into paragraphs, vertical lists are broken into pages, and formulas are built out of math lists. Also the output to the `dvi` file takes place on this level. The algorithms working here are not accessible to the user, but they can be influenced by a number of parameters.

2 The input processor

The input processor is that part of TEX that translates whatever characters it gets from the input file into tokens. The output of this processor is a stream of tokens: a token list. Most tokens fall into two categories: character tokens and control sequence tokens. The remaining category is that of the parameter tokens; these will not be treated here.

[1] To be published in TUGboat, © 1991, TEX Users Group.
[2] This is a chapter from my book 'TEX by Topic', to be published by Addison-Wesley.
[3] Tokens being 'sicked up again' [2], output being 'TEXcrement' [3], or the particularly deplorable title of [4] . . .

2.1 Character input

For simple input text, characters are made into character tokens. However, TEX can ignore input characters: a row of spaces in the input is usually equivalent to just one space. Also, TEX itself can insert tokens that do not correspond to any character in the input, for instance the space token at the end of an input line, or the `\par` token after an empty line.

Not all character tokens represent characters that are to be typeset. Characters fall into sixteen categories – each one specifying a certain function that a character can have – of which only two contain the characters that will be typeset. The other categories contain such characters as {, }, `&`, and `#`. A character token can be considered as a pair of numbers: the character code – usually the ASCII code – and the category code.

When the escape character \ appears in the input, TEX's behaviour in forming tokens is more complicated. Basically, TEX builds a control sequence by taking a number of characters from the input and lumping them together into a single token.

The behaviour with which TEX's input processor reacts to category codes can be described as finite-state automaton with three internal states: N, new line, M, middle of line, and S, skipping spaces. These states and the transitions between them are treated in chapter 8 of *The TEXbook*.

2.2 Two-level input processing

TEX's input processor is in fact even a two-level processor. Due to limitations of the terminal, the editor, or the operating system, the user may not be able to input any desired character. Therefore, TEX provides a mechanism to access with two superscript characters all of the available character positions. This may be considered to be a separate stage of TEX processing, taking place prior to the three-state finite automaton mentioned above.

For instance, the sequence `^^+` is replaced by `k` because the ASCII codes of `k` and `+` differ by 64. Since this replacement takes place before tokens are formed, one may write `\vs^^+ip 5cm` to get the effect of `\vskip 5cm`. More useful examples than this exist.

Note that this first stage is a transformation from characters to characters, without considering category codes. These come into play only in the second phase of input processing, where characters are converted to character tokens by coupling the category code to the character code.

3 The expansion processor

TEX's expansion processor accepts a stream of tokens and, if possible, expands the tokens in this stream one by one until only unexpandable tokens remain. Macro expansion is the clearest example of this: if a control sequence is a macro name, it is replaced (together possibly with parameter tokens) by the definition text of the macro.

Input for the expansion processor is provided mainly by the input processor. The stream of tokens coming from the first stage of TEX processing is subject to the expansion process, and the result is a stream of unexpandable tokens which is fed to the execution processor.

However, the expansion processor comes into play also when an `\edef` or `\write` is processed. The parameter token list of these commands is expanded as if the lists would have been on top level, instead of the argument to a command.

There is a special fascination to macros that work completely by the expansion processor. See the recent articles [4], [5], and [6] for some good examples.

3.1 The process of expansion

Expanding a token comprises the following steps:

- See if the token is expandable.
- If the token is unexpandable, pass it to the token list currently being built, and take on the next token.
- If the token is expandable, replace it by its expansion. For macros without parameters, and a few primitive commands such as `\jobname`, this is indeed a simple replacement. Usually, however, TEX needs to absorb some argument tokens from the stream in order to be able to form the replacement of the current token. For instance, if the token was a macro with parameters, sufficiently many tokens need to be absorbed to form the arguments corresponding to these parameters.
- Go on expanding, starting with the first token of the expansion.

Deciding whether a token is expandable is usually a simple decision. Macros and active characters, conditionals, and a number of primitive TEX commands (see the list on page 215 of *The TEXbook*) are expandable, other tokens are not. Thus the expansion processor replaces macros by their expansion, it evaluates conditionals and eliminates any irrelevant parts of these, but tokens such as `\vskip` and character tokens, including characters such as dollars and braces, are passed untouched.

3.2 Special cases: `\expandafter`, `\noexpand`, and `\the`

As stated above, after a token has been expanded TEX will start expanding the resulting tokens. At first sight the `\expandafter` command would seem to be an exception to this rule, because it expands only one step. What actually happens is that the sequence

`\expandafter`<token$_1$><token$_2$>

is replaced by

$<\text{token}_1><\textit{expansion of token}_2>$

and this replacement is in fact reexamined by the expansion processor.

Real exceptions do exist, however. If the current token is the `\noexpand` command, the next token is considered for the moment to be unexpandable: it is handled as if it were `\relax` (more about this control sequence follows below), and it is passed to the token list being built.

Example: in the macro definition

```
\edef\a{\noexpand\b}
```

the replacement text `\noexpand\b` is expanded at definition time. The expansion of `\noexpand` is the next token, with a temporary meaning of `\relax`. Thus, when the expansion processor tackles the next token, the `\b`, it will consider that to be unexpandable, and just pass it to the token list being built, which is the replacement text of the macro.

Another exception is that the tokens resulting from `\the`<token variable> are not expanded further if this statement occurs inside an `\edef` macro definition.

3.3 Braces in the expansion processor

Above, it was said that braces are passed as unexpandable character tokens. In general this is true. For instance, the `\romannumeral` command is handled by the expansion processor; when confronted with

```
\romannumeral1\number\count2 3{4
```

TeX will expand until the brace is encountered: if `\count2` has the value of zero, the result will be the roman numeral representation of `103`.

As another example,

```
\iftrue {\else }\fi
```

is handled by the expansion processor as if it were

```
\iftrue a\else b\fi
```

The result is a character token, be this a brace or a letter.

However, in the context of macro expansion the expansion processor will recognize braces. First of all, a balanced pair of braces marks off a group of tokens to be passed as one argument. If a macro has an argument

```
\def\macro#1{ ... }
```

one can call it with a single token

```
\macro 1 \macro \$
```

or with a group

```
\macro {abc} \macro {d{ef}g}
```

Secondly, when the arguments for a macro with parameters are read, no expressions with unbalanced braces are accepted. In

```
\def\a#1\stop{ ... }
\a bc{d\stop}e\stop
```

the argument is `bc{d\stop}e`. Only balanced expressions are accepted here.

4 The execution processor

The execution processor builds lists: horizontal, vertical, and math lists. Corresponding to these lists, it works in horizontal, vertical, or math mode. Of these three modes ‘internal’ and ‘external’ variants exist. In addition to building lists, this part of the TeX processor also performs mode-independent processing, such as assignments.

Coming out of the expansion processor is a stream of unexpandable tokens to be processed by the execution processor. From the point of view of the execution processor, this stream contains two types of tokens:

- Tokens that signal an assignment (this includes macro definitions), and other tokens that are independent of the mode, such as `\show` and `\aftergroup`.
- Tokens that build lists: characters, boxes, and glue.

Some objects can be used in any mode, for instance boxes can appear in horizontal, vertical, and math lists. The effect of such an object will of course still depend on the mode. Other objects are specific for one mode. For instance, characters (to be more precise: character tokens of categories 11 and 12) are intimately connected to horizontal mode: if the execution processor is in vertical mode when it encounters a character, it will switch to horizontal mode.

For the expansion processor a character token is just an unexpandable object. On this level, however, something is actually done with it. Some characters are typeset, but the execution processor can also encounter, for instance, math shift characters (usually `$`), or braces. When a math shift character is found in the stream of tokens, math mode is entered (or exited if the current mode was math mode); when a left brace is found, a new level of grouping is entered.

One control sequence handled by the execution processor deserves special mention: `\relax`. This control sequence is not expandable, but the execution is ‘empty’. Compare the effect of `\relax` in

```
\count0=1\relax 2
```

with that of `\null` defined by

```
\def\null{}
```

in

```
\count0=1\null 2
```

In the first case the expansion process that is forming the number stops at `\relax` because it is unexpandable, and the number `1` is assigned. In the second case `\null` expands to nothing, so `12` is assigned.

5 The visual processor

TEX's visual processor encompasses those algorithms that are outside direct user control: paragraph breaking, alignment, page breaking, math typesetting, and `dvi` file generation. Various parameters control the operation of these parts of TEX.

Some of these algorithms return their results in a form that can be handled by the execution processor. For instance, a paragraph that has been broken into lines is added to the main vertical list as a sequence of horizontal boxes with intermediate glue and penalties. Also, the page breaking algorithm stores its result in `\box255`, so output routines can disect it. On the other hand, a math formula can not be broken into pieces, and, of course, shipping a box to the `dvi` file is irreversible.

6 Further examples

6.1 Skipped spaces

Skipped spaces provide an illustration of the view that TEX's levels of processing accept the completed input of the previous level. Consider the commands

```
\def\a{\penalty200}
\a 0
```

Faulty reasoning

"The `\a` is encountered, expanded, the space then delimits the number"

would lead to the conclusion that this is equivalent to `\penalty200 0`. It is not. Instead, what results is

```
\penalty2000
```

because the space after `\a` is skipped in the input processor.

6.2 Internal quantities and their representations

TEX uses various sorts of internal quantities, such as integers and dimensions. These internal quantities have an external representation, which is a string of characters, such as `4711` or `91.44cm`.

Conversions between the internal value and the external represenation take place on two different levels, depending on the direction the conversion goes. A string of characters is converted to an internal value in assignments such as

```
\pageno=12 \baselineskip=13pt
```

or statements like

```
\vskip 5.71pt
```

and all of these statements are handled by the execution processor.

On the other hand, the conversion of the internal values into a representation as a string of characters is handled by the expansion processor. For instance,

```
\number\pageno \romannumeral\year
\the\baselineskip
```

are all processed by expansion.

Note that in the `\baselineskip` example above the conversion from string of characters to internal value was 'automatic'. The conversion the other way has to be forced by a command such as `\number`. Thus there is no danger that the sequence

```
\pageno=3 \count\MyCount=\pageno 5
```

will result in assigning either 15 or 35 to `\MyCount`.

As a final example, suppose `\count2=45`, and consider the statement

```
\count0=1\number\count2 3
```

The expansion processor tackles `\number\count2` to give the characters `45`, and the space after the `2` is absorbed because it only serves as a delimiter of the number of the `\count` register. In the next stage of processing, the execution processor will then see the statement

```
\count0=1453
```

and execute this.

References

[1] Donald Knuth, *The TEXbook,*. Addison–Wesley Publishing Company, 1984.

[2] Angela Barden, Some TEX manuals, TUGboat 12(1991), no. 1.

[3] Ron Whitney, private communication.

[4] Victor Eijkhout, Oral TEX, TUGboat 12(1991), no. 2.

[5] Alan Jeffrey, Lists in TEX's mouth, TUGboat 11, no. 2, 237–245.

[6] Sonja Maus, Looking ahead for a ⟨box⟩, TUGboat 11, no. 4, 613–614.