Low-level: Tokens

Hans Hagen

1. Introduction

Most users don't need to know anything about tokens but when TEXies meet in person (user group meetings) or online (support platforms), some folk always seem to pop-up and speak about tokens. When you try to explain something to a user, it makes sense to talk in terms of characters, but soon those token-speakers jump in and start correcting you. In the past I have been puzzled by this because when one can write a decent macro that does the job well, it really doesn't matter if one knows about tokens. Of course one should never make the assumption that token-speakers really know TEX that well or can come up with better solutions than users but that is another matter.[1]

That said, since the word 'token' does pop-up in documents about TEX, I will try to give you some insights even if it's mostly irrelevant when using TEX. The descriptions below won't match the proper token-speak criteria for sure. This is why at the presentation for the 2020 user meeting, I used the title "Tokens As I See Them."

2. What are tokens?

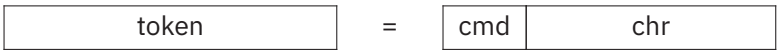
Both the words 'node' and 'token' are quite common in programming and also rather old, proven by the fact that they are also used in the TEX source. A node is a storage container that is part of a linked list. When you input the characters `t`, `e` and `x`, the three characters become part of the current linked list. They become 'character' nodes (or in LuaTEX speak 'glyph' nodes) with properties like the font and the reference character. But before this happens, the three characters in the input (`t`, `e` and `x`) are interpreted as just that: characters. When you enter `\TeX`, the input processors first sees the backslash and because this has a special meaning in TEX, it will read the following characters and return when done to lookup the internal hash table. In this case, a macro that assembled the word TEX in uppercase with special kerning and a shifted (therefore boxed) 'E'. When you enter `$`, TEX will look ahead for a second one in order to determine whether to enter display math mode, push back the found token when there is no match and then enter inline math mode.

A token is internally just a 32-bit number that encodes what TEX has seen. It is the assembled token that travels through the system, gets stored, interpreted and often discarded afterwards. So, the character 'e' in our example gets tagged as such and

[1] Talking about fashion: it would be more impressive to talk about TEX and friends as a software stack rather than as a distribution. Today, it's all about marketing.

encoded in a 32-bit number in a way that the intention can be derived later on.

Now, the way TEX looks at these tokens can vary. In some cases, it will just look at the 32-bit number (for instance when checking for a specific token; which is a fast operation) but other times, it needs to know some more details. The mentioned integer actually encodes a command (opcode) and a so-called 'char code' (operand). The second name is somewhat confusing because in many cases the char code does not represent a character (although this is not relevant here). When you look at the source code of a TEX engine, it is enough to know that a char can also be a sub command.



Back to the three characters: these become tokens where the command code indicates that it is a letter and the char code indicates what letter we have at hand; and in the case of LuaTEX and LuaMetaTEX these are Unicode values. Contrary to the traditional 8-bit TEX engine, in the Unicode engines an UTF sequence is read, but these multiple bytes still become one number that will be encoded in the token number. In order to determine that something is a letter, the engine has to be told that it is one (which is what a macro package does when it sets up the engine). For instance, digits are so-called 'other characters' and the backslash is called 'escape'. Every TEX user knows that curly braces are special, and so are dollar symbols and hashes. If this rings a bell, and you relate this to catcodes, you can indeed assume that the command codes of these tokens have the same numbers as the catcodes. Given that Unicode has plenty of character slots you can imagine that combining 16 catcode commands with all the possible Unicode values makes a large repertoire of tokens.

There are more commands than the 16 basic character-related ones. In LuaMetaTEX we have just over 150 command codes and LuaTEX has a few more but they are also organized differently. Each of these codes can have a sub-command. For instance, the primitives `\vbox` and `\hbox` are both a `make_box_cmd` (we use the symbolic name here) and in LuaMetaTEX the first one has sub-command code 9 (`vbox_code`) and the second one has code 10 (`hbox_code`). There are twelve primitives in the same sub-command category. The many primitives that make up the core of the engine are grouped in a way that permits the processing of similar ones with one function, and also makes it possible to distinguish between the way commands are handled, for instance with respect to expansion.

Now, before we move on, it is important to know that all these codes are in fact abstract numbers. Although it is quite likely that engines that are derived from each other have similar numbers (just more), this is not the case for LuaMetaTEX. Because the internals have been opened up (even more than in LuaTEX), the command and char codes have been reorganized in a such a way that the exposure is consistent. We could not use some of the reuse and remap tricks that the other engines use because it would simply be too confusing (and demand real in-depth knowledge of the internals). This is also the reason why development has taken some time. You

probably won't notice it from the current source code but it was a very stepwise process. We not only had to make sure that it all kept working (ConTEXt LMTX and LuaMetaTEX were pretty useable during the process), but we also had to (re)consider intermediate choices.

So, input is converted into tokens, and in most cases one-by-one. When a token is assembled, it either gets stored (deliberately or as part of some look ahead scanning), or it immediately gets (what is called) 'expanded'. Depending on what the command is, some action is triggered. For instance, a character gets appended to the node list immediately. An `\hbox` command will start assembling a box with its own node list that is then processed. If the primitive was a follow-up on `\setbox` it would get stored, otherwise it might end up in the current node list as a so-called hlist node. Commands that relate to registers have `0xFFFF` char codes because that is how many registers we have per category.

When a token gets stored for later processing, it becomes part of a larger data structure, a so called 'memory word'. These memory words store a token and some additional properties, and are taken from a large pool of such memory words. The 'info' field contains the token value, the aforementioned command and char. When there is no linked list, the link can be used to store a value; something that we actually do in LuaMetaTEX.

1	info	link
2	info	link
3	info	link
n	info	link

When, for instance, we say `\toks 0 {tex}`, the scanner sees an escape followed by 4 letters (`toks`) and the escape triggers a lookup of the primitive (or macro or `...`) with that name; in this case, a primitive assignment command. The found primitive (its property gets stored in the token) triggers scanning for a number and when that is successful, scanning of a brace delimited token list starts. The three characters become three-letter tokens, which are linked lists of the aforementioned memory words. This list then gets stored in token register zero. The input sequence `\the \toks 0` will return a copy of this list back into the input.

In addition to the token memory pool, there is also a table of equivalents. This one is part of a larger table of memory words where TEX stores everything it needs to keep. The 16 groups of character commands are virtual. Storing these makes no sense so the first real entries are all the registers (count, dimension, skip, box, etc.). The rest is taken up by possible hash entries.

main hash	null control sequence
	128K hash entries
	frozen control sequences
	special sequences (undefined)
registers	17 internal & 64K user glues
	4 internal & 64K user mu glues
	12 internal & 64K user tokens
	2 internal & 64K user boxes
	116 internal & 64K user integers
	0 internal & 64K user attribute
	22 internal & 64K user dimensions
specifications	5 internal & 0 user
extra hash	additional entries (grows dynamically)

So, a letter token `t` is just that, a token. A token referring to a register is again just a number, but its char code points to a slot in the equivalents table. A macro, which we haven't discussed yet, is actually just a token list. When a name lookup happens, the hash table is consulted and this table has parallel entries in the table of equivalents. When there is a match, the corresponding entry in the equivalents table will point to a token list.

1	string index	equivalents or (next > n) index
2	string index	equivalents or (next > n) index
n	string index	equivalents or (next > n) index
n + 1	string index	equivalents or (next > n) index
n + 2	string index	equivalents or (next > n) index
n + m	string index	equivalents or (next > n) index

It sounds complex but it is actually somewhat complex. It is not made easier by the fact that we also track information related to grouping (saving and restoring), that we need reference counts for copies of macros and token lists, and that sometimes we need to store information directly instead of via links to token lists, etc. And again, we cannot compare LuaMetaTEX with the other engines. Since we did away with some of the limitations of the traditional engine, we not only saved some memory but in the end we also simplified matters (we're 32/64-bit after all). On the one hand, while some traditional speedups have been removed, these have been compensated for by improvements elsewhere, making the overall processing more efficient.

1	level	type	flag	value
2	level	type	flag	value
3	level	type	flag	value
n	level	type	flag	value

So, here LuaMetaTEX differs from other engines because it combines two tables, which is made possible because we have at least 32 bits. There are at most `0xFFFF` levels but we need at most `0xFF` types. In LuaMetaTEX, macros can have extra properties (flags) and these also need one byte. Contrary to the other engines, `\protected` macros are native and have their own command code, but `\tolerant` macros duplicate that, so we have four distinct macro commands. All other properties, like the `\permanent` ones are stored in flags.

Because a macro starts with a reference count, we have some room in the info field to store information about it, whether the macro has arguments or not. It is these details that make LuaMetaTEX a bit more efficient in terms of memory usage and performance than its predecessor LuaTEX. But as with the other changes, it was a stepwise process in order to keep the system compatible and working.

3. Some implementation details

Sometimes there is a special head token at the start of the linked list to make it easier to append extra tokens. In traditional TEX, node lists are forward linked; in LuaTEX they are double linked[2]. Token lists are always forward linked, and shared token lists use the head node for a reference count.

For various reasons the original TEX uses temporary lists of global variables. This is, for instance, needed when we expand (nested) and need to report issues. But in LuaTEX we often just serialize lists, so using local variables makes more sense. One of the first things done in LuaMetaTEX was to group all global variables into (still global) structures, albeit well isolated ones. This also made it possible to actually remove some globals.

Because TEX was designed to run on machines that we would nowadays consider rather limited, it had to be sparse and efficient. There are quite a few optimizations implemented to limit code and memory consumption. The engine also does its own memory management. Freed token memory words are collected in a cache and reused, but they can get scattered. This is not too bad although it may adversely affect cache hits. In LuaMetaTEX, we stay as close to original TEX as possible but there are some improvements. The Lua interfaces force us to occasionally divert from the original design. This might, in fact, lead to some retrofitting but the original documentation still mostly applies. However, keep in mind that in LuaTEX, we store much more information in nodes than traditional TEX does. Each has a prev pointer,

[2] On the agenda of LuaMetaTEX is to use this property in the underlying code.

an attribute list pointer and some other additional fields; for instance, glyph nodes have some 20 extra fields compared to traditional TEX character nodes.

4. Other data management

There is plenty going on in TEX when it processes your input. Just to mention a few:

- Grouping is handled by a nesting stack.
- Nested conditionals (`\if...`) have their own stack.
- The values before assignments are saved on the save stack.
- Also other local changes (housekeeping) ends up in the save stack.
- Token lists and macro aliases have references pointers (reuse).
- Attributes, being linked node lists, have their own management.

In all these subsystems, tokens or references to tokens can play a role. Reading a single character from the input can trigger a lot of action. A curly brace tagged as a 'begin group command' will be pushed to the grouping level. From then registers and some other quantities that have changed will be stored on the save stack so that after the group ends, these quantities can be restored. When primitives take keywords, and no match happens, tokens are pushed back into the input which introduces a new input level (also some stack). When numbers are read, a token that represents no digit is pushed back too. Macro packages use numbers and dimensions extensively so it is a surprise that TEX is so fast.

5. Macros

There is a distinction between primitives, the built-in commands, and macros (the commands defined by users). A primitive relates to a command code and char code but macros are basically pointers to a token list, unless they are made an alias to something else like `\countdef` and `\let` do. There is some additional data stored which makes it possible to parse and grab arguments.

When we have a control sequence (macro) `\controlsequence` the name is looked up in the hash table. When it is found, its value will point to the table of equivalents. As mentioned, that table keeps track of the cmd and points to a token list (the meaning). We saw that this table also stores the current level of grouping and flags.

If we say in the input, `\hbox to 10pt {x\hss}`, the box is assembled as the tokens are processed, and when it is appended to the current node list, there are no tokens left to process. When scanning this input, the engine literally sees a backslash and the four letters `hbox`. However, when we have this:

```
\def\MyMacro{\hbox to 10pt {x\hss}}
```

the `\hbox` has become one memory word which has a token representing the `\hbox` primitive plus a link to the next token. The space after a control sequence is gobbled so the next two tokens, again stored in a linked memory word, are letter-tokens

followed by two others and two letter-tokens for the dimensions. Then we have a space, a brace, a letter, a primitive and a brace. The approximately 20 characters of input became a dozen memory words, each two times four bytes so in terms of memory usage, we end up with quite a bit more. However, when TEX runs over that list, it only has to interpret the token values because the scanning and conversion have already happened. So, the space that a macro takes is more than compensated for by the efficient reprocessing.

6. Looking at tokens

When you use the `\tracingall` command, you will see what the engine does: read input, expand primitives and macros, typesetting etc. You might need to set `\tracingonline` to get a bit more output on the console. One way to look at macros is to use the `\meaning` command, so if we have:

```
\permanent\protected\def\MyMacro#1#2{Do #1 or #2!}
```

we can say this:

```
\meaning    \MyMacro
\meaningless\MyMacro
\meaningfull\MyMacro
```

and get:

```
protected  macro:#1#2->Do #1 or #2!
#1#2->Do #1 or #2!
permanent protected  macro:#1#2->Do #1 or #2!
```

You get just the name when you ask for the meaning of a primitive. The `\meaningfull` primitive gives the most information. In LuaMetaTEX protected macros are first class commands: they have their own command code. In other engines, they are just regular macros with an initial token indicating that they are protected. There are specific command codes for `\outer` and `\long` macros but we dropped these in LuaMetaTEX. Instead we have `\tolerant` macros but this is another story. The flags that were mentioned earlier can mark macros in a way that permits overload protection, as well as permit special treatment for some otherwise tricky cases (like alignments). The overload related flags permit a rather granular way of preventing users from redefining macros and such. They are set via prefixes, and add to that repertoire, we have 14 prefixes, only eight of which deal with flags (we can add more if really needed). The probably most well-known prefix is `\global`, and this one will never become a flag: it has immediate effect.

For the above definition, the `\showluatokens` command will show a meaning on the console.

```
\showluatokens\MyMacro
```

This gives the next list, where the first column is the address of the token, the second one is the command code, and the third one is the char code. When there are arguments involved, the list of what needs to get matched is shown.

```
permanent protected control sequence: MyMacro
501263  19   49  match                argument 1
501087  19   50  match                argument 2
385528  20    0  end match
--------------
501090  11   68  letter               D (U+00044)
 30833  11  111  letter               o (U+0006F)
500776  10   32  spacer
385540  21    1  parameter reference
112057  10   32  spacer
431886  11  111  letter               o (U+0006F)
 30830  11  114  letter               r (U+00072)
 30805  10   32  spacer
500787  21    2  parameter reference
213412  12   33  other char           ! (U+00021)
```

In the next subsections, I will show some examples. This time we use a helper defined in the `system-tokens` module:

```
\usemodule[system-tokens]
```

6.1 Example 1: in the input

```
\luatokentable{1 \bf{2} 3\what {!}}
```

given token list:						
654818	12	49	other char	1	U+00031	
655581	10	32	spacer			
652047	132	0	protected call			bf
656462	1	123	left brace			
654849	12	50	other char	2	U+00032	
653177	2	125	right brace			
654476	10	32	spacer			
654446	12	51	other char	3	U+00033	
654394	119	0	undefined cs			what
653170	1	123	left brace			
653887	12	33	other char	!	U+00021	
655416	2	125	right brace			

6.2 Example 2: in the input

```
\luatokentable{a \the\scratchcounter b \the\parindent \hbox to
10pt{x}}
```

given token list:						
654519	11	97	letter	a	U+00061	
652401	10	32	spacer			
654938	129	0	the			the
654316	85	257	register int			scratchcounter
652731	11	98	letter	b	U+00062	
655070	10	32	spacer			
652654	129	0	the			the
656364	88	0	internal dimen			parindent
653328	30	10	make box			hbox
654483	11	116	letter	t	U+00074	
652859	11	111	letter	o	U+0006F	
654238	10	32	spacer			
652657	12	49	other char	1	U+00031	
656366	12	48	other char	0	U+00030	
652014	11	112	letter	p	U+00070	
653119	11	116	letter	t	U+00074	
655095	1	123	left brace			
654115	11	120	letter	x	U+00078	
652370	2	125	right brace			

6.3 Example 3: user registers

```
\scratchtoks{foo \framed{\red 123}456}

\luatokentable\scratchtoks
```

token register: scratchtoks						
653239	11	102	letter	f	U+00066	
651553	11	111	letter	o	U+0006F	
653011	11	111	letter	o	U+0006F	
652400	10	32	spacer			
650534	135	0	tolerant protected call			framed
654522	1	123	left brace			
653666	132	0	protected call			red
654939	12	49	other char	1	U+00031	
655001	12	50	other char	2	U+00032	
652733	12	51	other char	3	U+00033	
655742	2	125	right brace			
656485	12	52	other char	4	U+00034	
656245	12	53	other char	5	U+00035	
291878	12	54	other char	6	U+00036	

6.4 Example 4: internal variables

```
\luatokentable\everypar
```

internal token variable: everypar				
652275	132	0	protected call	dotagsetparcounter
653066	132	0	protected call	page_otr_command_synchronize_side_floats
652527	132	0	protected call	checkindentation
653213	131	0	call	showparagraphnumber
653786	132	0	protected call	restoreinterlinepenalty

653728	131	0	call	flushnotes
654553	132	0	protected call	registerparoptions
654003	131	0	call	flushpostponednodedata
656318	131	0	call	typo_delimited_repeat
653835	131	0	call	spac_paragraphs_flush_intro
655388	131	0	call	typo_initial_handle
656894	131	0	call	typo_firstline_handle
653683	131	0	call	spac_paragraph_wrap
656402	132	0	protected call	spac_paragraph_freeze

6.5 Example 5: macro definitions

```
\protected\def\whatever#1[#2](#3)\relax
  {oeps #1 and #2 & #3 done ## error}

\luatokentable\whatever
```

protected control sequence: whatever						
654851	19	49	match		argument 1	
656549	12	91	other char	[	U+0005B	
651965	19	50	match		argument 2	
654812	12	93	other char	]	U+0005D	
654903	12	40	other char	(	U+00028	
652258	19	51	match		argument 3	
651906	12	41	other char	)	U+00029	
651871	16	0	relax			relax
652010	20	0	end match			
651468	11	111	letter	o	U+0006F	
655309	11	101	letter	e	U+00065	
651491	11	112	letter	p	U+00070	
652656	11	115	letter	s	U+00073	
654837	10	32	spacer			
150496	21	1	parameter reference			
652586	10	32	spacer			
652111	11	97	letter	a	U+00061	
655292	11	110	letter	n	U+0006E	
652838	11	100	letter	d	U+00064	
651998	10	32	spacer			
653176	21	2	parameter reference			
655491	10	32	spacer			
656546	12	38	other char	&	U+00026	
652018	10	32	spacer			
658198	21	3	parameter reference			
655541	10	32	spacer			
655383	11	100	letter	d	U+00064	
654373	11	111	letter	o	U+0006F	
656288	11	110	letter	n	U+0006E	
655762	11	101	letter	e	U+00065	
653339	10	32	spacer			
652800	6	35	parameter			
651511	10	32	spacer			
652693	11	101	letter	e	U+00065	
655195	11	114	letter	r	U+00072	
654332	11	114	letter	r	U+00072	

652764	11	111	letter	o	U+0006F
655939	11	114	letter	r	U+00072

6.6 Example 6: commands

```
\luatokentable\startitemize
```

frozen instance protected control sequence: startitemize					
652854	135	0	tolerant protected call		startitemgroup
658207	12	91	other char	[	U+0005B
655907	11	105	letter	i	U+00069
654234	11	116	letter	t	U+00074
654137	11	101	letter	e	U+00065
653596	11	109	letter	m	U+0006D
653468	11	105	letter	i	U+00069
653159	11	122	letter	z	U+0007A
654444	11	101	letter	e	U+00065
653908	12	93	other char	]	U+0005D

6.7 Example 7: commands

```
\luatokentable\doifelse
```

permanent protected control sequence: doifelse					
653143	19	49	match	argument 1	
656590	19	50	match	argument 2	
656234	20	0	end match		
658288	126	21	if test		iftok
653842	1	123	left brace		
652602	21	1	parameter reference		
651850	2	125	right brace		
652769	1	123	left brace		
652549	21	2	parameter reference		
655517	2	125	right brace		
656196	120	0	expand after		expandafter
653605	131	0	call		firstoftwoarguments
656657	126	3	if test		else
655972	120	0	expand after		expandafter
652924	131	0	call		secondoftwoarguments
655224	126	2	if test		fi

6.8 Example 8: nothing

```
\luatokentable\relax
```

primitive control sequence: relax				
652599	16	0	relax	relax

6.9 Example 9: hashes

```
\edef\foo#1#2{(#1)(\letterhash)(#2)}  \luatokentable\foo
```

control sequence: foo					
655352	19	49	match		argument 1
653257	19	50	match		argument 2
655054	20	0	end match		
652976	12	40	other char	(	U+00028
651999	21	1	parameter reference		
654246	12	41	other char	)	U+00029
654954	12	40	other char	(	U+00028
656553	12	35	other char	#	U+00023
654494	12	41	other char	)	U+00029
655274	12	40	other char	(	U+00028
655845	21	2	parameter reference		
653351	12	41	other char	)	U+00029

6.10 Example 10: nesting

```
\def\foo#1{\def\foo##1{(#1)(##1)}}  \luatokentable\foo
```

control sequence: foo						
656526	19	49	match		argument 1	
653870	20	0	end match			
654395	115	1	def			def
655739	131	0	call			foo
652667	6	35	parameter			
654579	12	49	other char	1	U+00031	
658014	1	123	left brace			
651678	12	40	other char	(	U+00028	
653618	21	1	parameter reference			
655191	12	41	other char	)	U+00029	
656774	12	40	other char	(	U+00028	
652487	6	35	parameter			
654320	12	49	other char	1	U+00031	
657716	12	41	other char	)	U+00029	
653247	2	125	right brace			

In all these examples, the numbers are to be seen as abstractions. Some command codes and sub-command codes might change as the engine evolves. This is why the LuaMetaTEX engine has lots of Lua functions that provide information about which number represents what command.