

low level

TEX

tokens

## Contents

1	Introduction	1
2	What are tokens	1
3	Some implementation details	5
4	Other data management	6
5	Macros	7
6	Looking at tokens	7

## 1 Introduction

Most users don't need to know anything about tokens but it happens that when T<sub>E</sub>Xies meet in person (users group meetings), or online (support platforms) there always seem to pop up folks who love token speak. When you try to explain something to a user it makes sense to talk in terms of characters but then those token speakers can 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 T<sub>E</sub>X that well or can come up with better solutions than users but that is another matter.<sup>1</sup>

That said, because in documents about T<sub>E</sub>X the word 'token' does pop up I will try to give a little insight here. But for using T<sub>E</sub>X it's mostly irrelevant. The descriptions below for sure won't match the proper token speak criteria which is why at a 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 which is proven by the fact that they also are used in the T<sub>E</sub>X source. A node is a storage container that is part of a linked list. When you input the characters tex the three characters become part of the current linked list. They become 'character' nodes (or in LuaT<sub>E</sub>X speak 'glyph' nodes) with properties like the font and the character referred to. But before that happens, the three characters in the input t, e and x, are interpreted as in this case being just that: characters. When you enter \TeX the input processors first sees a backslash and because that has a special meaning in T<sub>E</sub>X it will

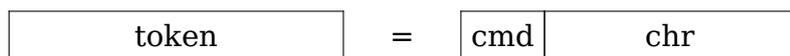
---

<sup>1</sup> Talking about fashion: it would be more impressive to talk about T<sub>E</sub>X and friends as a software stack than calling it a distribution. Today, it's all about marketing.

read following characters and when done does a lookup in it's internal hash table to see what it actually is: 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 display math, 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, get stored, interpreted and often discarded afterwards. So, the character 'e' in our example gets tagged as such and encoded in this number in a way that the intention can be derived later on.

Now, the way  $\TeX$  looks at these tokens can differ. In some cases it will just look at this (32 bit) number, for instance when checking for a specific token, which is fast, but sometimes it needs to know some detail. 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 that code is not representing a character but that is not that 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 stores what letter we have at hand and in the case of  $\text{Lua}\TeX$  and  $\text{LuaMeta}\TeX$  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 (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 characters 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 characters related ones, in  $\text{LuaMeta}\TeX$  we have just over 150 command codes ( $\text{Lua}\TeX$  has a few more but they are also organized differently). Each of these codes can have a sub command, For instance the primitives  $\backslash\text{vbox}$  and  $\backslash\text{hbox}$  are both a `make_box_cmd` (we use the symbolic name here) and in  $\text{LuaMeta}\TeX$  the first one has sub command code 9 (`vbox_code`) and the second one has code 10 (`hbox_code`). There are twelve primitives that are in the same category.

## What are tokens

The many primitives that make up the core of the engine are grouped in a way that permits processing 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 LuaMetaT<sub>E</sub>X. Because the internals have been opened up (even more than in LuaT<sub>E</sub>X) the command and char codes have been reorganized in a such a way that 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 took some time. You probably won't notice it from the current source but it was a very stepwise process. We not only had to make sure that all kept working (ConT<sub>E</sub>Xt LMTX and LuaMetaT<sub>E</sub>X were pretty useable during the process), but also had to (re)consider intermediate choices.

So, input is converted into tokens, 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 which its own node list that then gets some treatment: if this primitive was a follow up on `\setbox` it will get stored, otherwise it might end up in the current node list as 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 are taken from a large pool of words and they store a token and additional properties. The info field contains the token value, the mentioned command and char. When there is no linked list, the link can actually be used to store a value, something that in LuaMetaT<sub>E</sub>X we actually do.

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 and these are a linked list of the mentioned memory words. This list then gets stored in token register zero. The input sequence `\the\toks 0` will push back a copy of this list into the input.

In addition to the token memory pool, there is also a table of equivalents. That one is part of a larger table of memory words where  $\text{\TeX}$  stores all it needs to store. The 16 groups of character commands are virtual, storing these makes no sense, so the first real entries are all these 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 dynamic)

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 that one runs in parallel to part of the table of equivalents. When there is a match, the corresponding entry in the equivalents table points 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 and it actually also is somewhat complex. It is not made easier by the fact that we also track information related to grouping (saving and restoring), need reference counts for copies of macros and token lists, sometimes store information directly instead of via links to token lists, etc. And again one cannot compare LuaMetaTeX with the other engines. Because we did away with some of the limitations of the traditional engine we not only could save some memory but in the end also simplify matters (we're 32/64 bit after all). On the one hand some traditional speedups were removed but these have been compensated by improvements elsewhere, so overall processing is 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 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 the flags.

Because a macro starts with a reference count we have some room in the info field to store information about it having arguments or not. It is these details that make LuaMetaTeX a bit more efficient in terms of memory usage and performance than its ancestor LuaTeX. But as with the other changes, it was a very 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. This makes for easier appending of extra tokens. In traditional TeX node lists are forward linked, in LuaTeX they are

double linked<sup>2</sup>. Token lists are always forward linked. Shared token lists use the head node for a reference count.

For various reasons original T<sub>E</sub>X uses global variables temporary lists. This is for instance needed when we expand (nested) and need to report issues. But in LuaT<sub>E</sub>X we often just serialize lists and using local variables makes more sense. One of the first things done in LuaMetaT<sub>E</sub>X was to group all global variables in (still global) structures but well isolated. That also made it possible to actually get rid of some globals.

Because T<sub>E</sub>X had to run on machines that we nowadays consider rather limited, it had to be sparse and efficient. There are quite some optimizations 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 which is not that bad, apart from maybe cache hits. In LuaMetaT<sub>E</sub>X we stay as close to original T<sub>E</sub>X as possible but there have been some improvements. The Lua interfaces force us to occasionally divert from the original, and that in fact might lead to some retrofit but the original documentation still mostly applies. However, keep in mind that in LuaT<sub>E</sub>X we store much more in nodes (each has a prev pointer and an attribute list pointer and for instance glyph nodes have some 20 extra fields compared to traditional T<sub>E</sub>X character nodes).

## 4 Other data management

There is plenty going on in T<sub>E</sub>X 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 begin group command will push the grouping level and from then on registers and some other quantities that are changed will be stored on the save stack so that after the group ends they 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).

---

<sup>2</sup> On the agenda of LuaMetaT<sub>E</sub>X is to use this property in the underlying code, that doesn't yet profit from this and therefore keep previous pointers in store.

When numbers are read a token that represents no digit is pushed back too and macro packages use numbers and dimensions a lot. It is a surprise that T<sub>E</sub>X is so fast.

## 5 Macros

There is a distinction between primitives, the build in commands, and macros, the commands defined by users. A primitive relates to a command code and char code but macros are, unless they are made an alias to something else, like a `\countdef` or `\let` does, basically pointers to a token list. There is some additional data stored that 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 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 we go and when it is appended to the current node list there are no tokens left. When scanning this, 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 other and two letter tokens for the dimensions. Then we have a space, a brace, a letter, a primitive and a brace. The about 20 characters in the 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 T<sub>E</sub>X runs over that list it only has to interpret the token values because the scanning and conversion already happened. So, the space that a macro takes is more than compensated by efficient reprocessing.

## 6 Looking at tokens

When you say `\tracingall` 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 less when you ask for the meaning of a primitive, just its name. The `\meaningfull` primitive gives the most information. In LuaMetaTeX protected macros are first class commands: they have their own command code. In the 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 that in LuaMetaTeX. Instead we have `\tolerant` macros but that's another story. The flags that were mentioned can mark macros in a way that permits overload protection as well as some special treatment in otherwise tricky cases (like alignments). The overload related flags permits a rather granular way to prevent users from redefining macros and such. They are set via prefixes, and add to that repertoire: we have 14 prefixes but only some eight deal with flags (we can add more if really needed). The probably most well known prefix is `\global` and that one will not 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 the command code, and the third one 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 give some examples. This time we use helper defined in a module:

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

## 6.1 Example 1: in the input

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

---

### given token list:

543282	12	49	other char	1	U+00031	
32650	10	32	spacer			
538531	138	0	protected call			bf
32747	1	123	left brace			
543108	12	50	other char	2	U+00032	
203842	2	125	right brace			
542992	10	32	spacer			
543165	12	51	other char	3	U+00033	
158531	125	0	undefined cs			what
542910	1	123	left brace			
540378	12	33	other char	!	U+00021	
540484	2	125	right brace			

---

## 6.2 Example 2: in the input

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

---

### given token list:

540332	11	97	letter	a	U+00061	
543406	10	32	spacer			
542953	135	0	the			the
542736	108	123	integer			scratchcounter
543123	11	98	letter	b	U+00062	
539153	10	32	spacer			
542853	135	0	the			the
540401	90	0	internal dimen			parindent
543045	30	14	make box			hbox
542965	11	116	letter	t	U+00074	
543341	11	111	letter	o	U+0006F	
32716	10	32	spacer			

539282	12	49	other char	1	U+00031
540389	12	48	other char	0	U+00030
32727	11	112	letter	p	U+00070
226978	11	116	letter	t	U+00074
211396	1	123	left brace		
543301	11	120	letter	x	U+00078
542868	2	125	right brace		

---

### 6.3 Example 3: user registers

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

```
\luatokenable\scratchtoks
```

---

#### token register: scratchtoks

---

560486	11	102	letter	f	U+00066	
542767	11	111	letter	o	U+0006F	
543352	11	111	letter	o	U+0006F	
543224	10	32	spacer			
542766	141	0	tolerant protected call			framed
32667	1	123	left brace			
32668	138	0	protected call			red
543054	12	49	other char	1	U+00031	
540445	12	50	other char	2	U+00032	
211381	12	51	other char	3	U+00033	
539352	2	125	right brace			
543376	12	52	other char	4	U+00034	
543280	12	53	other char	5	U+00035	
542745	12	54	other char	6	U+00036	

---

### 6.4 Example 4: internal variables

```
\luatokenable\everypar
```

---

#### internal token variable: everypar

---

540495	138	0	protected call	dotagsetparcounter	
540346	138	0	protected call	page_otr_command_synchronize_side_floats	
542916	138	0	protected call	checkindentation	
119138	137	0	call	showparagraphnumber	
179407	138	0	protected call	restoreinterlinepenalty	
543226	137	0	call	flushnotes	
543231	138	0	protected call	registerparoptions	
172030	137	0	call	flushpostponednodedata	
211385	137	0	call	typo_delimited_repeat	
226971	137	0	call	spac_paragraphs_flush_intro	
540503	137	0	call	typo_initial_handle	
542814	137	0	call	typo_firstline_handle	

211377	137	0	call	spac_paragraph_wrap
540417	138	0	protected call	spac_paragraph_freeze

---

## 6.5 Example 5: macro definitions

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

```
\luatokenable\whatever
```

---

### protected control sequence: whatever

---

32749	19	49	match		argument 1
211416	12	91	other char	[	U+0005B
542821	19	50	match		argument 2
542781	12	93	other char	]	U+0005D
543216	12	40	other char	(	U+00028
543190	19	51	match		argument 3
543402	12	41	other char	)	U+00029
542699	16	0	relax		relax
538403	20	0	end match		
<hr/>					
542648	11	111	letter	o	U+0006F
543106	11	101	letter	e	U+00065
539332	11	112	letter	p	U+00070
540379	11	115	letter	s	U+00073
543167	10	32	spacer		
539340	21	1	parameter reference		
543189	10	32	spacer		
32674	11	97	letter	a	U+00061
32714	11	110	letter	n	U+0006E
32750	11	100	letter	d	U+00064
87878	10	32	spacer		
540473	21	2	parameter reference		
543117	10	32	spacer		
542966	12	38	other char	&	U+00026
543124	10	32	spacer		
542717	21	3	parameter reference		
542661	10	32	spacer		
540293	11	100	letter	d	U+00064
158532	11	111	letter	o	U+0006F
543040	11	110	letter	n	U+0006E
542982	11	101	letter	e	U+00065
543459	10	32	spacer		
543056	6	35	parameter		
543111	10	32	spacer		
32729	11	101	letter	e	U+00065
261633	11	114	letter	r	U+00072
542634	11	114	letter	r	U+00072
543178	11	111	letter	o	U+0006F
539356	11	114	letter	r	U+00072

---

## 6.6 Example 6: commands

`\luatokentable\startitemize`

`\luatokentable\stopitemize`

---

### frozen instance protected control sequence: startitemize

---

543121	141	0	tolerant protected call		startitemgroup
543295	12	91	other char	[ U+0005B	
539174	11	105	letter	i U+00069	
540462	11	116	letter	t U+00074	
543105	11	101	letter	e U+00065	
32700	11	109	letter	m U+0006D	
542619	11	105	letter	i U+00069	
540372	11	122	letter	z U+0007A	
542825	11	101	letter	e U+00065	
543156	12	93	other char	] U+0005D	

---

### frozen instance protected control sequence: stopitemize

---

543125	138	0	protected call	stopitemgroup
--------	-----	---	----------------	---------------

---

## 6.7 Example 7: commands

`\luatokentable\doifelse`

---

### permanent protected control sequence: doifelse

---

540525	19	49	match	argument 1
542700	19	50	match	argument 2
540274	20	0	end match	
543183	132	26	if test	iftok
542605	1	123	left brace	
540377	21	1	parameter reference	
540211	2	125	right brace	
538562	1	123	left brace	
543390	21	2	parameter reference	
385132	2	125	right brace	
32680	126	0	expand after	expandafter
543407	137	0	call	firstoftwoarguments
543148	132	3	if test	else
32701	126	0	expand after	expandafter
542822	137	0	call	secondoftwoarguments
539266	132	2	if test	fi

---

## 6.8 Example 8: nothing

`\luatokentable\relax`

---

**primitive control sequence: relax**

---

543458	16	0	relax	relax
--------	----	---	-------	-------

---

## 6.9 Example 9: hashes

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

---

**control sequence: foo**

---

540529	19	49	match	argument 1
539173	19	50	match	argument 2
32736	20	0	end match	
543323	12	40	other char	( U+00028
540545	21	1	parameter reference	
540555	12	41	other char	) U+00029
542762	12	40	other char	( U+00028
543289	12	35	other char	# U+00023
543322	12	41	other char	) U+00029
542564	12	40	other char	( U+00028
542956	21	2	parameter reference	
543096	12	41	other char	) U+00029

---

## 6.10 Example 10: nesting

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

---

**control sequence: foo**

---

540563	19	49	match	argument 1
539374	20	0	end match	
542769	121	1	def	def
539298	137	0	call	foo
540195	6	35	parameter	
539296	12	49	other char	1 U+00031
539337	1	123	left brace	
543452	12	40	other char	( U+00028
540276	21	1	parameter reference	
543011	12	41	other char	) U+00029
539421	12	40	other char	( U+00028
538526	6	35	parameter	
32720	12	49	other char	1 U+00031
539215	12	41	other char	) U+00029
544591	2	125	right brace	

---

## 6.11 Remark

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 Lua-MetaT<sub>E</sub>X engine has lots of Lua functions that provide information about what number represents what command.

## 6.11 Colofon

Author	Hans Hagen
ConT <sub>E</sub> Xt	2023.04.27 17:04
LuaMetaT <sub>E</sub> X	2.1008
Support	<a href="http://www.pragma-ade.com">www.pragma-ade.com</a> <a href="http://contextgarden.net">contextgarden.net</a>