

---

## L<sup>A</sup>T<sub>E</sub>X3: An outsider's overview

Joseph Wright

### Abstract

The current experimental L<sup>A</sup>T<sub>E</sub>X3 packages provide a new, documented programming interface for T<sub>E</sub>X. The key ideas implemented in this new interface are highlighted in this article.

### 1 Introduction

Modifying the behaviour of L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> often requires a combination of user macros, internal L<sup>A</sup>T<sub>E</sub>X macro and T<sub>E</sub>X primitives. This makes even trial modifications of document layout potentially difficult, even for the experienced L<sup>A</sup>T<sub>E</sub>X user. The differing syntax used by T<sub>E</sub>X primitives and the L<sup>A</sup>T<sub>E</sub>X kernel only add to the confusion here.

The first step to develop a new L<sup>A</sup>T<sub>E</sub>X kernel is therefore to address how the underlying system is programmed. Rather than the current mix of L<sup>A</sup>T<sub>E</sub>X and T<sub>E</sub>X macros, the experimental L<sup>A</sup>T<sub>E</sub>X3 system provides its own consistent interface to all of the functions needed to control T<sub>E</sub>X. A key part of this work is to ensure that everything is documented, so that L<sup>A</sup>T<sub>E</sub>X users can work efficiently without needing to be familiar with the internal nature of the kernel or with plain T<sub>E</sub>X.

The current kernel also suffers from the mixing of design commands with structural code. Thus changing a layout element often requires modifying a kernel code block (or loading a package which provides an interface to achieve this). The second challenge for L<sup>A</sup>T<sub>E</sub>X3 is therefore separation of the basic tools of the kernel from the design of documents.

This short overview article highlights the key developments to date in L<sup>A</sup>T<sub>E</sub>X3. It is based on my own experience working with the new tools for writing packages, and a talk given recently to the UK T<sub>E</sub>X Users Group.

### 2 The components of L<sup>A</sup>T<sub>E</sub>X3

Currently, the experimental L<sup>A</sup>T<sub>E</sub>X3 packages are designed to be used “on top of” L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>. This avoids needing to wait for the entire kernel to be finished before testing what is written.

The most developed part of the code is the `expl3` bundle, the core of the new kernel providing the new programming interface. The new language is fully documented in the file `source3.pdf`, which contains some notes for the experienced (L<sup>A</sup>)T<sub>E</sub>X programmer.

Built on top of `expl3` is the `xparse` package. This is meant to be a “bridge” between the internal and user parts of the new kernel. The `xparse` package

is used to create new user macros, in a much more controlled way than is possible using `\newcommand`.

More experimental than `xparse` are various other “`xpackages`”. These are designed to explore new approaches to layout and document design for L<sup>A</sup>T<sub>E</sub>X3.

The most complete part of L<sup>A</sup>T<sub>E</sub>X3 is the `expl3` bundle. The rest of this article is focussed mainly on the new internal syntax introduced in `expl3`.

### 3 A new internal syntax

L<sup>A</sup>T<sub>E</sub>X3 does not use `@` as a “letter” for defining internal macros. Instead, the symbols `_` and `:` are used in internal macro names to provide structure. In contrast to the plain T<sub>E</sub>X format and the L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> kernel, these extra letters are used only between parts of a macro name (no strange vowel replacement).

L<sup>A</sup>T<sub>E</sub>X3 separates macros which do something (functions) from ones which only store data. The general form of an internal function in L<sup>A</sup>T<sub>E</sub>X3 is `\<module>_<function>:<arg-spec>`. The `<module>` prefix is applied to almost all macros. For a package, it will typically be the package name; the kernel is split into a number of modules, each with its own name. The name of the `<function>` should give a good description of what it does: this may contain one or more `_` characters to divide the name into logical units.

The concept of the `<arg-spec>` is potentially confusing to existing (L<sup>A</sup>)T<sub>E</sub>X programmers. This *argument specifier* describes the arguments expected by the function. In most cases, each argument is represented by a single letter. The letter and its case then conveys information about the type of argument required. The use of the `<arg-spec>` is illustrated later in this article.

#### 3.1 Primitives renamed

All of the T<sub>E</sub>X primitives are given new names by `expl3`, although many are not intended to be used outside of the L<sup>A</sup>T<sub>E</sub>X3 kernel. Instead, a number of L<sup>A</sup>T<sub>E</sub>X wrappers for primitives are provided, so that the argument syntax is consistent.

At the most basic level, the `\fi` primitive becomes `\fi:`, indicating that no arguments are required. A more complex example is `\ifx`, which becomes `\if_meaning:wN`.

```
\if_meaning:wN \Macro_One \Macro_Two
% Do Stuff
\fi
```

Here, the `<arg-spec>` contains two letters, showing that two arguments are required. The first argument here is `w`, meaning “weird”, as there are rather exacting requirements for the first token in the comparison.

The second argument is of type N, meaning that it should be single token *not* surrounded by braces.

### 3.2 Example kernel functions

Renaming primitives helps to keep the new syntax consistent, but does not show why the argument specifier is useful. This is perhaps best seen by looking at some of the functions provided by `expl3`.

By using the argument specifier, the new kernel provides families of related functions which avoid the need for complex `\expandafter` runs. For example, the TeX primitive `\let` can only be used with two macro names. In L<sup>A</sup>T<sub>E</sub>X<sub>3</sub>, the family of `\let`-like macros contains:

```
\cs_set_eq:NN \Macro_One \Macro_Two
\cs_set_eq:Nc \Macro_One {Macro_Two}
\cs_set_eq:cN {Macro_One} \Macro_Two
\cs_set_eq:cc {Macro_One} {Macro_Two}
```

where the argument specified as `c` be given in braces and should expand to a `csname`. This is much clearer than the equivalent plain TeX constructions; taking `\cs_set_eq:Nc` as an example:

```
\expandafter\let\expandafter\Macro_One
\csname Macro_Two\endcsname.
```

The specifiers `n` (no expansion), `o` (expand once) and `x` (`\edef`-like expansion) allow large families of related functions to be created easily, so that using the results is easy. Thus we can create a macro `\Macro_One:nn`, then create `\Macro_One:no`, `\Macro_One:xn` and so on very rapidly. Later, we will see how the `v` and `V` argument specifiers add even more power to this concept.

The argument specifier concept also makes testing much easier. As an example, the new kernels provides three tests related to the `\ifundefined` macro:

```
\cs_if_exist:cT {csname} {true}
\cs_if_exist:cF {csname} {false}
\cs_if_exist:cTF {csname} {true} {false}
```

In all three cases, the first argument will be converted to a `csname` (the `c` specifier). The first two functions then require one more argument, either `T` or `F`. As might be expected, these are executed if the test is true or false, respectively. The third function (ending `:cTF`) has both a true and false branch. By providing tests with the choice of `T`, `F` and `TF` arguments, empty groups in code can be avoided and meaning is much more obvious.

## 4 Data storage

In L<sup>A</sup>T<sub>E</sub>X<sub>3</sub>, macros which carry out some process are called functions, and all contain an argument specifier. Macros used for storage are handled separately,

to help to make code cleaner and easier to read. To further aid the programmer, `expl3` defines several new data types:

- Token list pointers (`tlp`);
- Comma lists (`clist`);
- Property lists (`prop`);
- Sequences (`seq`).

in addition to the existing types, which are renamed:

- Boolean switches (`bool`);
- Counters (`int`);
- Skips (`skip`);

and so on.

The name “token list pointer” may cause confusion, and so some background is useful. TeX works with tokens and lists of tokens, rather than characters. It provides two ways to store these token lists, within macros and as token registers (`toks`). L<sup>A</sup>T<sub>E</sub>X<sub>3</sub> retains the name “`toks`” for the later, and adopts the name token list pointer for macros used to store tokens. In most circumstances, the `tlp` data type is more convenient for storing token lists.

The other new variable types are all essentially lists of items separated by a special token. The nature of the separator determines the type of variable and what functions apply. For example, a comma list is, rather obviously, a set of tokens separated by commas.

These are all created explicitly as either local or global. For example, a `tlp` may be named

```
\l_<module>_<name>_tlp
```

(local) or

```
\g_<module>_<name>_tlp
```

(global). The other variable types follow the same pattern, with the appropriate type identified in the variable name.

As well as the new data types, `expl3` provides a range of functions for manipulating data. Often, these have to have been coded by hand when using L<sup>A</sup>T<sub>E</sub>X<sub>2 $\epsilon$</sub> . For example, `\tlp_elt_count:N` is available, to count the number of elements (often characters) in a `tlp`.

## 5 Expanding variables

When coding in (L<sup>A</sup>)TeX, the need to access data in variables is made more complicated by the different possibilities for recovering information later. For example, if three macros are defined as

```
\def\tempa{Some text}
\def\tempb{\tempa}
\def\tempc{\tempb}
```

then there are two likely scenarios for using the information in `\tempc`:

- Use of the value that `\tempc` contains (in this case `\tempb`);
- Exhaustive expansion of `\tempc` to use the unexpandable token list it represents (in this case “Some text”).

The situation is further complicated as macros do not need an accessor function, whereas other  $\TeX$  variables (toks, counts, skips) do. This leads to the need for carefully-constructed `\expandafter` runs in  $(\LaTeX)$ , in order to get the content the programmer intends.

To avoid this,  $\LaTeX$ 3 provides two argument specifiers which will always return the content of a variable. The `V` specifier requires the name of a variable, and returns the content. For example, if we define two variables

```
\toks_set:Nn \l_my_toks { Text \mymacro }
\tlp_set:Nn \l_my_tlp { Text \mymacro }
```

and pass them to some function `\foo_bar:V`

```
\foo_bar:V \l_my_toks
\foo_bar:V \l_my_tlp
```

both sets of input will result in “Text `\mymacro`” being passed as the argument to the underlying function `\foo_bar:n`. The `V` specifier also works in the same way with other  $\LaTeX$ 3 variables.

The second “variable” specifier is `v`. This converts its argument to a `csname`, then recovers the content of the resulting variable and passes the content. Thus we might have `\foo_bar:v` and use it as

```
\foo_bar:v { l_my_toks }
\foo_bar:v { l_my_tlp }
```

with the same result as the previous example.

The two variable specifiers are very powerful. By using them, the programmer can avoid almost entirely the need to worry about the order of expansion when using stored information.

## 6 Other key features

The new kernel will require the  $\varepsilon$ - $\TeX$  extensions. This means that the new primitives are definitely available when working with  $\LaTeX$ 3. For example, `\unexpanded` is part of the expansion module, as `\exp_not:n`.

Boolean switches in  $\TeX$  and  $\LaTeX$ 2 $\varepsilon$  use the `\iftrue` and `\iffalse` primitives. This can lead to problems with nesting (Incomplete `\if...`). to avoid this,  $\LaTeX$ 3 does not create switches in the same way. This means that all of the switches use

exclusively  $\LaTeX$  syntax, and require an “access” function.

```
\bool_if:NT \l_example_bool { true code }
\bool_if:NF \l_example_bool { false code }
\bool_if:NTF \l_example_bool { true code }
{ false code }
```

One of the most useful features of the new coding syntax is the treatment of white space. The literal space character ( ) is ignored inside code block, meaning that the text can be laid out to aid ease of reading. When a space is required in the output, the hard space (`~`) is used. The ability to finish lines without needing `%` is highly welcome!

## 7 Conclusions

The current  $\LaTeX$ 3 modules provide a new and powerful programming language for  $\TeX$ . The full details of the language are collected in one place, and the language is much more logical than the current mix of  $\TeX$  and  $\LaTeX$ 2 $\varepsilon$ .  $\LaTeX$ 3 is therefore ready for serious use by  $(\LaTeX)$  programmers.

At this stage, the document level of  $\LaTeX$ 3 is much less defined. It seems likely that good separation of programming and document design will be made available. The new code syntax means that a number of ideas currently implemented as independent packages will need to be re-implemented either in the new kernel or as supported tools.

My own experience with  $\LaTeX$ 3 convinces me that the kernel team need outsiders to use the code. The team have done a very good job so far, but everyone will bring new approaches using to the code. With the involvement of the wider  $\TeX$  community,  $\LaTeX$ 3 has the potential to be a major step forward for  $\LaTeX$ .

◊ Joseph Wright  
Morning Star  
2, Dowthorpe End  
Earls Barton  
Northampton NN6 0NH  
United Kingdom  
joseph.wright@morningstar2.co.  
uk