The expl3 package and LaTeX3 programming

The LaTeX3 Project[*]

Released 2017/11/14

Abstract

This document gives an introduction to a new set of programming conventions that have been designed to meet the requirements of implementing large scale TeX macro programming projects such as LaTeX. These programming conventions are the base layer of LaTeX3.

The main features of the system described are:

- classification of the macros (or, in LaTeX terminology, commands) into LaTeX functions and LaTeX parameters, and also into modules containing related commands;
- a systematic naming scheme based on these classifications;
- a simple mechanism for controlling the expansion of a function's arguments.

This system is being used as the basis for TeX programming within the LaTeX3 project. Note that the language is not intended for either document mark-up or style specification. Instead, it is intended that such features will be built on top of the conventions described here.

This document is an introduction to the ideas behind the expl3 programming interface. For the complete documentation of the programming layer provided by the LaTeX3 Project, see the accompanying `interface3` document.

1 Introduction

The first step to develop a LaTeX kernel beyond LaTeX 2_ε is to address how the underlying system is programmed. Rather than the current mix of LaTeX and TeX macros, the LaTeX3 system provides its own consistent interface to all of the functions needed to control TeX. A key part of this work is to ensure that everything is documented, so that LaTeX programmers and users can work efficiently without needing to be familiar with the internal nature of the kernel or with plain TeX.

The expl3 bundle provides this new programming interface for LaTeX. To make programming systematic, LaTeX3 uses some very different conventions to LaTeX 2_ε or plain TeX. As a result, programmers starting with LaTeX3 need to become familiar with the syntax of the new language.

The next section shows where this language fits into a complete TeX-based document processing system. We then describe the major features of the syntactic structure of command names, including the argument specification syntax used in function names.

The practical ideas behind this argument syntax will be explained, together with the expansion control mechanism and the interface used to define variant forms of functions.

[*]E-mail: latex-team@latex-project.org

As we shall demonstrate, the use of a structured naming scheme and of variant forms for functions greatly improves the readability of the code and hence also its reliability. Moreover, experience has shown that the longer command names which result from the new syntax do not make the process of *writing* code significantly harder.

2 Languages and interfaces

It is possible to identify several distinct languages related to the various interfaces that are needed in a TEX-based document processing system. This section looks at those we consider most important for the LATEX3 system.

Document mark-up This comprises those commands (often called tags) that are to embedded in the document (the `.tex` file).

It is generally accepted that such mark-up should be essentially *declarative*. It may be traditional TEX-based mark-up such as LATEX 2ε, as described in [3] and [2], or a mark-up language defined via HTML or XML.

One problem with more traditional TEX coding conventions (as described in [1]) is that the names and syntax of TEX's primitive formatting commands are ingeniously designed to be "natural" when used directly by the author as document mark-up or in macros. Ironically, the ubiquity (and widely recognised superiority) of logical mark-up has meant that such explicit formatting commands are almost never needed in documents or in author-defined macros. Thus they are used almost exclusively by TEX programmers to define higher-level commands, and their idiosyncratic syntax is not at all popular with this community. Moreover, many of them have names that could be very useful as document mark-up tags were they not pre-empted as primitives (*e.g.* `\box` or `\special`).

Designer interface This relates a (human) typographic designer's specification for a document to a program that "formats the document". It should ideally use a declarative language that facilitates expression of the relationship and spacing rules specified for the layout of the various document elements.

This language is not embedded in document text and it will be very different in form to the document mark-up language. For LATEX, this level was almost completely missing from LATEX2.09; LATEX 2ε made some improvements in this area but it is still the case that implementing a design specification in LATEX requires far more "low-level" coding than is acceptable.

Programmer interface This language is the implementation language within which the basic typesetting functionality is implemented, building upon the primitives of TEX (or a successor program). It may also be used to implement the previous two languages "within" TEX, as in the current LATEX system.

The last layer is covered by the conventions described in this document, which describes a system aimed at providing a suitable basis for coding LATEX3. Its main distinguishing features are summarised here:

- A consistent naming scheme for all commands, including TEX primitives.
- The classification of commands as LATEX functions or LATEX parameters, and also their division into modules according to their functionality.

- A simple mechanism for controlling argument expansion.
- Provision of a set of core LaTeX functions that is sufficient for handling programming constructs such as queues, sets, stacks, property lists.
- A TeX programming environment in which, for example, all white space is ignored.

3 The naming scheme

LaTeX3 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 TeX format and the LaTeX 2_ε kernel, these extra letters are used only between parts of a macro name (no strange vowel replacement).

While TeX is actually a macro processor, by convention for the `expl3` programming language we distinguish between *functions* and *variables*. Functions can have arguments and they are either expanded or executed. Variables can be assigned values and they are used in arguments to functions; they are not used directly but are manipulated by functions (including getting and setting functions). Functions and variables with a related functionality (for example accessing counters, or manipulating token lists, *etc.*) are collected together into a *module*.

3.1 Examples

Before giving the details of the naming scheme, here are a few typical examples to indicate the flavour of the scheme; first some variable names.

`\l_tmpa_box` is a local variable (hence the `l_` prefix) corresponding to a box register.
`\g_tmpa_int` is a global variable (hence the `g_` prefix) corresponding to an integer register (i.e. a TeX count register).
`\c_empty_tl` is the constant (`c_`) token list variable that is always empty.

Now here is an example of a typical function name.

`\seq_push:Nn` is the function which puts the token list specified by its second argument onto the stack specified by its first argument. The different natures of the two arguments are indicated by the `:Nn` suffix. The first argument must be a single token which "names" the stack parameter: such single-token arguments are denoted `N`. The second argument is a normal TeX "undelimited argument", which may either be a single token or a balanced, brace-delimited token list (which we shall here call a *braced token list*): the `n` denotes such a "normal" argument form. The name of the function indicates it belongs to the `seq` module.

3.2 Formal naming syntax

We shall now look in more detail at the syntax of these names. A function name in LaTeX3 has a name consisting of three parts:

\⟨*module*⟩_⟨*description*⟩:⟨*arg-spec*⟩

while a variable has (up to) four distinct parts to its name:

\⟨*scope*⟩_⟨*module*⟩_⟨*description*⟩_⟨*type*⟩

The syntax of all names contains

⟨*module*⟩ and ⟨*description*⟩

these both give information about the command.

A *module* is a collection of closely related functions and variables. Typical module names include `int` for integer parameters and related functions, `seq` for sequences and `box` for boxes.

Packages providing new programming functionality will add new modules as needed; the programmer can choose any unused name, consisting of letters only, for a module. In general, the module name and module prefix should be related: for example, the kernel module containing `box` functions is called `l3box`. Module names and programmers' contact details are listed in l3prefixes.csv.

The *description* gives more detailed information about the function or parameter, and provides a unique name for it. It should consist of letters and, possibly, `_` characters. In general, the description should use `_` to divide up "words" or other easy to follow parts of the name. For example, the LaTeX3 kernel provides `\if_cs_exist:N` which, as might be expected, tests if a command name exists.

Where functions for variable manipulation can perform assignments either locally or globally, the latter case is indicated by the inclusion of a `g` in the second part of the function name. Thus `\tl_set:Nn` is a local function but `\tl_gset:Nn` acts globally. Functions of this type are always documented together, and the scope of action may therefore be inferred from the presence or absence of a `g`. See the next subsection for more detail on variable scope.

3.2.1 Separating private and public material

One of the issues with the TeX language is that it doesn't support name spaces and encapsulation other than by convention. As a result nearly every internal command in the LaTeX 2_ε kernel has eventually be used by extension packages as an entry point for modifications or extensions. The consequences of this is that nowadays it is next to impossible to change anything in the LaTeX 2_ε kernel (even if it is clearly just an internal command) without breaking something.

In expl3 we hope to improve this situation drastically by clearly separating public interfaces (that extension packages can use and rely on) and private functions and variables (that should not appear outside of their module). There is (nearly) no way to enforce this without severe computing overhead, so we implement it only through a naming convention, and some support mechanisms. However, we think that this naming convention is easy to understand and to follow, so that we are confident that this will adopted and provides the desired results.

Functions created by a module may either be "public" (documented with a defined interface) or "private" (to be used only within that module, and thus not formally documented). It is important that only documented interfaces are used; at the same time, it is necessary to show within the name of a function or variable whether it is public or private.

To allow clear separation of these two cases, the following convention is used. Private functions should be defined with `__` added to the beginning of the module name. Thus

```
\module_foo:nnn
```

is a public function which should be documented while

```
\__module_foo:nnn
```

is private to the module, and should *not* be used outside of that module.

In the same way, private variables should use two `__` at the start of the module name, such that

```
\l_module_foo_tl
```

is a public variable and

```
\l__module_foo_tl
```

is private.

3.2.2 Using `@@` and l3docstrip to mark private code

The formal syntax for internal functions allows clear separation of public and private code, but includes redundant information (every internal function or variable includes `__`⟨*module*⟩). To aid programmers, the l3docstrip program introduces the syntax

```
    %<@@=⟨module⟩>
```

which then allows `@@` (and `_@@` in case of variables) to be used as a place holder for `__`⟨*module*⟩ in code. Thus for example

```
%<@@=foo>
%    \begin{macrocode}
\cs_new:Npn \@@_function:n #1
  ...
\tl_new:N \l_@@_my_tl
%    \end{macrocode}
```

is converted by l3docstrip to

```
\cs_new:Npn \__foo_function:n #1
  ...
\tl_new:N \l__foo_my_tl
```

on extraction. As you can see both `_@@` and `@@` are mapped to `__`⟨*module*⟩, because we think that this helps to distinguish variables from functions in the source when the `@@` convention is used.

3.2.3 Variables: scope and type

The ⟨*scope*⟩ part of the name describes how the variable can be accessed. Variables are classified as local, global or constant. This *scope* type appears as a code at the beginning of the name; the codes used are:

c constants (global variables whose value should not be changed);

g variables whose value should only be set globally;

l variables whose value should only be set locally.

Separate functions are provided to assign data to local and global variables; for example, `\tl_set:Nn` and `\tl_gset:Nn` respectively set the value of a local or global "token list" variable. Note that it is a poor TeX practice to intermix local and global assignments to a variable; otherwise you risk exhausting the save stack.[1]

The ⟨*type*⟩ is in the list of available *data-types*;[2] these include the primitive TeX data-types, such as the various registers, but to these are added data-types built within the LaTeX programming system.

The data types in LaTeX3 are:

bool either true or false (the LaTeX3 implementation does not use `\iftrue` or `\iffalse`);

box box register;

clist comma separated list;

coffin a "box with handles" — a higher-level data type for carrying out `box` alignment operations;

dim "rigid" lengths;

fp floating-point values;

ior an input stream (for reading from a file);

iow an output stream (for writing to a file);

int integer-valued count register;

muskip math mode "rubber" lengths;

prop property list;

seq sequence: a data-type used to implement lists (with access at both ends) and stacks;

skip "rubber" lengths;

str TeX strings: a special case of `tl` in which all characters have category "other" (catcode 12), other than spaces which are category "space" (catcode 10);

tl "token list variables": placeholders for token lists.

When the ⟨*type*⟩ and ⟨*module*⟩ are identical (as often happens in the more basic modules) the ⟨*module*⟩ part is often omitted for aesthetic reasons.

The name "token list" 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`). The implementation in LaTeX3 means that `toks` are not required, and that all operations for storing tokens can use the `tl` variable type.

Experienced TeX programmers will notice that some of the variable types listed are native TeX registers whilst others are not. In general, the underlying TeX implementation for a data structure may vary but the *documented interface* will be stable. For example, the `prop` data type was originally implemented as a `toks`, but is currently built on top of the `tl` data structure.

[1]See *The TeXbook*, p. 301, for further information.

[2]Of course, if a totally new data type is needed then this will not be the case. However, it is hoped that only the kernel team will need to create new data types.

3.2.4 Variables: guidance

Both comma lists and sequences have similar characteristics. They both use special delimiters to mark out one entry from the next, and are both accessible at both ends. In general, it is easier to create comma lists 'by hand' as they can be typed in directly. User input often takes the form of a comma separated list and so there are many cases where this is the obvious data type to use. On the other hand, sequences use special internal tokens to separate entries. This means that they can be used to contain material that comma lists cannot (such as items that may themselves contain commas!). In general, comma lists should be preferred for creating fixed lists inside programs and for handling user input where commas will not occur. On the other hand, sequences should be used to store arbitrary lists of data.

expl3 implements stacks using the sequence data structure. Thus creating stacks involves first creating a sequence, and then using the sequence functions which work in a stack manner (`\seq_push:Nn`, *etc.*).

Due to the nature of the underlying TEX implementation, it is possible to assign values to token list variables and comma lists without first declaring them. However, this is *not supported behavior*. The LATEX3 coding convention is that all variables must be declared before use.

The expl3 package can be loaded with the `check-declarations` option to verify that all variables are declared before use. This has a performance implication and is therefore intended for testing during development and not for use in production documents.

3.2.5 Functions: argument specifications

Function names end with an ⟨*arg-spec*⟩ after a colon. This gives an indication of the types of argument that a function takes, and provides a convenient method of naming similar functions that differ only in their argument forms (see the next section for examples).

The ⟨*arg-spec*⟩ consists of a (possibly empty) list of letters, each denoting one argument of the function. The letter, including its case, conveys information about the type of argument required.

All functions have a base form with arguments using one of the following argument specifiers:

n Unexpanded token or braced token list.
This is a standard TEX undelimited macro argument.

N Single token (unlike `n`, the argument must *not* be surrounded by braces).
A typical example of a command taking an `N` argument is `\cs_set`, in which the command being defined must be unbraced.

p Primitive TEX parameter specification.
This can be something simple like `#1#2#3`, but may use arbitrary delimited argument syntax such as: `#1,#2\q_stop#3`. This is used when defining functions.

T,F These are special cases of `n` arguments, used for the true and false code in conditional commands.

There are two other specifiers with more general meanings:

D This means: **Do not use**. This special case is used for TEX primitives. Programmers outside the kernel team should not use these functions!

w This means that the argument syntax is "weird" in that it does not follow any standard rule. It is used for functions with arguments that take non standard forms: examples are TeX-level delimited arguments and the boolean tests needed after certain primitive `\if`... commands.

In case of `n` arguments that consist of a single token the surrounding braces can be omitted in nearly all situations—functions that force the use of braces even for single token arguments are explicitly mentioned. However, programmers are encouraged to always use braces around `n` arguments, as this makes the relationship between function and argument clearer.

Further argument specifiers are available as part of the expansion control system. These are discussed in the next section.

4 Expansion control

Let's take a look at some typical operations one might want to perform. Suppose we maintain a stack of open files and we use the stack `\g_ior_file_name_seq` to keep track of them (`ior` is the prefix used for the file reading module). The basic operation here is to push a name onto this stack which could be done by the operation

```
\seq_gpush:Nn \g_ior_file_name_seq {#1}
```

where `#1` is the filename. In other words, this operation would push the file name as is onto the stack.

However, we might face a situation where the filename is stored in a variable of some sort, say `\l_ior_curr_file_tl`. In this case we want to retrieve the value of the variable. If we simply use

```
\seq_gpush:Nn \g_ior_file_name_seq \l_ior_curr_file_tl
```

we do not get the value of the variable pushed onto the stack, only the variable name itself. Instead a suitable number of `\exp_after:wN` would be necessary (together with extra braces) to change the order of expansion,[3] *i.e.*

```
\exp_after:wN
   \seq_gpush:Nn
\exp_after:wN
   \g_ior_file_name_seq
\exp_after:wN
   { \l_ior_curr_file_tl }
```

The above example is probably the simplest case but already shows how the code changes to something difficult to understand. Furthermore there is an assumption in this: that the storage bin reveals its contents after exactly one expansion. Relying on this means that you cannot do proper checking plus you have to know exactly how a storage bin acts in order to get the correct number of expansions. Therefore LaTeX3 provides the programmer with a general scheme that keeps the code compact and easy to understand.

To denote that some argument to a function needs special treatment one just uses different letters in the arg-spec part of the function to mark the desired behavior. In the above example one would write

[3] `\exp_after:wN` is the LaTeX3 name for the TeX `\expandafter` primitive.

```
\seq_gpush:NV \g_ior_file_name_seq \l_ior_curr_file_tl
```

to achieve the desired effect. Here the `V` (the second argument) is for "retrieve the value of the variable" before passing it to the base function.

The following letters can be used to denote special treatment of arguments before passing it to the base function:

c Character string used as a command name.
The argument (a token or braced token list) must, when fully expanded, produce a sequence of characters which is then used to construct a command name (*via* `\csname` ... `\endcsname`). This command name is the single token that is passed to the function as the argument. Hence

```
\seq_gpush:cV { g_file_name_seq } \l_tmpa_tl
```

is equivalent to

```
\seq_gpush:NV \g_file_name_seq \l_tmpa_tl.
```

Remember that `c` arguments are *fully expanded* by TEX when creating csnames. This means that (a) the entire argument must be expandable and (b) any variables are converted to their content. So the preceding examples are also equivalent to

```
\tl_new:N \g_file_seq_name_tl
\tl_gset:Nn \g_file_seq_name_tl { g_file_name_seq }
\seq_gpush:cV { \tl_use:N \g_file_seq_name_tl } \l_tmpa_tl.
```

(Token list variables are expandable and we could omit the accessor function `\tl_-use:N`. Other variable types require the appropriate `\<var>_use:N` functions to be used in this context.)

V Value of a variable.
This means that the contents of the register in question is used as the argument, be it an integer, a length-type register, a token list variable or similar. The value is passed to the function as a braced token list. Can be applied to variables which have a `\`⟨*var*⟩`_use:N` function, and which therefore deliver a single "value".

v Value of a register, constructed from a character string used as a command name.
This is a combination of `c` and `V` which first constructs a control sequence from the argument and then passes the value of the resulting register to the function. Can be applied to variables which have a `\`⟨*var*⟩`_use:N` function, and which therefore deliver a single "value".

x Fully-expanded token or braced token list.
This means that the argument is expanded as in the replacement text of an `\edef`, and the expansion is passed to the function as a braced token list. Expansion takes place until only unexpandable tokens are left. `x`-type arguments cannot be nested.

o One-level-expanded token or braced token list.
This means that the argument is expanded one level, as by `\expandafter`, and the expansion is passed to the function as a braced token list. Note that if the original argument is a braced token list then only the first token in that list is expanded. In general, using `V` should be preferred to using `o` for simple variable retrieval.

f Expanding the first token recursively in a braced token list.

Almost the same as the `x` type except here the token list is expanded fully until the first unexpandable token is found and the rest is left unchanged. Note that if this function finds a space at the beginning of the argument it gobbles it and does not expand the next token.

4.1 Simpler means better

Anyone who programs in TeX is frustratingly familiar with the problem of arranging that arguments to functions are suitably expanded before the function is called. To illustrate how expansion control can bring instant relief to this problem we shall consider two examples copied from `latex.ltx`.

```
\global\expandafter\let
      \csname\cf@encoding \string#1\expandafter\endcsname
      \csname ?\string#1\endcsname
```

This first piece of code is in essence simply a global `\let` whose two arguments firstly have to be constructed before `\let` is executed. The `#1` is a control sequence name such as `\textcurrency`. The token to be defined is obtained by concatenating the characters of the current font encoding stored in `\cf@encoding`, which has to be fully expanded, and the name of the symbol. The second token is the same except it uses the default encoding `?`. The result is a mess of interwoven `\expandafter` and `\csname` beloved of all TeX programmers, and the code is essentially unreadable.

Using the conventions and functionality outlined here, the task would be achieved with code such as this:

```
\cs_gset_eq:cc
  { \cf@encoding \token_to_str:N  #1 } { ? \token_to_str:N #1 }
```

The command `\cs_gset_eq:cc` is a global `\let` that generates command names out of both of its arguments before making the definition. This produces code that is far more readable and more likely to be correct first time. (`\token_to_str:N` is the LaTeX3 name for `\string`.)

Here is the second example.

```
\expandafter
  \in@
\csname sym#3%
  \expandafter
    \endcsname
  \expandafter
    {%
  \group@list}%
```

This piece of code is part of the definition of another function. It first produces two things: a token list, by expanding `\group@list` once; and a token whose name comes from '`sym#3`'. Then the function `\in@` is called and this tests if its first argument occurs in the token list of its second argument.

Again we can improve enormously on the code. First we shall rename the function `\in@`, which tests if its first argument appears within its second argument, according to our conventions. Such a function takes two normal "`n`" arguments and operates on token lists: it might reasonably be named `\tl_test_in:nn`. Thus the variant function we need would be defined with the appropriate argument types and its name would be `\tl_test_in:cV`. Now this code fragment would be simply:

```
\tl_test_in:cV { sym #3 } \group@list
```

This code could be improved further by using a sequence `\l_group_seq` rather than the bare token list `\group@list`. Note that, in addition to the lack of `\expandafter`, the space after the `}` is silently ignored since all white space is ignored in this programming environment.

4.2 New functions from old

For many common functions the LaTeX3 kernel provides variants with a range of argument forms, and similarly it is expected that extension packages providing new functions will make them available in all the commonly needed forms.

However, there will be occasions where it is necessary to construct a new such variant form; therefore the expansion module provides a straightforward mechanism for the creation of functions with any required argument type, starting from a function that takes “normal” TeX undelimited arguments.

To illustrate this let us suppose you have a “base function” `\demo_cmd:Nnn` that takes three normal arguments, and that you need to construct the variant `\demo_cmd:cnx`, for which the first argument is used to construct the *name* of a command, whilst the third argument must be fully expanded before being passed to `\demo_cmd:Nnn`. To produce the variant form from the base form, simply use this:

```
\cs_generate_variant:Nn \demo_cmd:Nnn { cnx }
```

This defines the variant form so that you can then write, for example:

```
\demo_cmd:cnx { abc } { pq } { \rst \xyz }
```

rather than ... well, something like this!

```
\def \tempa {{pq}}%
\edef \tempb {\rst \xyz}%
\expandafter
  \demo@cmd:nnn
\csname abc%
  \expandafter
    \expandafter
  \expandafter
      \endcsname
  \expandafter
    \tempa
  \expandafter
    {%
  \tempb
    }%
```

Another example: you may wish to declare a function `\demo_cmd_b:xcxcx`, a variant of an existing function `\demo_cmd_b:nnnnn`, that fully expands arguments 1, 3 and 5, and produces commands to pass as arguments 2 and 4 using `\csname`. The definition you need is simply

```
\cs_generate_variant:Nn \demo_cmd_b:nnnnn { xcxcx }
```

This extension mechanism is written so that if the same new form of some existing command is implemented by two extension packages then the two definitions are identical and thus no conflict occurs.

5 The distribution

At present, the expl3 modules are designed to be loaded on top of LaTeX 2$_\varepsilon$. In time, a LaTeX3 format will be produced based on this code. This allows the code to be used in LaTeX 2$_\varepsilon$ packages *now* while a stand-alone LaTeX3 is developed.

While expl3 is still experimental, the bundle is now regarded as broadly stable. The syntax conventions and functions provided are now ready for wider use. There may still be changes to some functions, but these will be minor when compared to the scope of expl3.

New modules will be added to the distributed version of expl3 as they reach maturity. At present, the expl3 bundle consists of a number of modules, most of which are loaded by including the line:

```
\RequirePackage{expl3}
```

in a LaTeX 2$_\varepsilon$ package, class or other file. The expl3 modules regarded as stable, and therefore suitable for basing real code on, are as follows:

l3basics This contains the basic definition modules used by the other packages.

l3box Primitives for dealing with boxes.

l3clist Methods for manipulating comma-separated token lists.

l3coffins Augmented box constructs for alignment operations.

l3expan This is the argument expansion module discussed earlier in this document.

l3int This implements the integer data-type `int`.

l3keys For processing lists of the form `{ key1=val1 , key2=val2 }`, intended to work as a LaTeX3 version of xkeyval/kvoptions, although with input syntax more like that of pgfkeys.

l3msg Communicating with the user: includes low-level hooks to allow messages to be filtered (higher-level interface for filtering to be written!).

l3names This sets up the basic naming scheme and renames all the TeX primitives.

l3prg Program control structures such as boolean data type `bool`, generic do-while loops, and conditional flow.

l3prop This implements the data-type for "property lists" that are used, in particular, for storing key/value pairs.

l3quark A "quark" is a command that is defined to expand to itself! Therefore they must never be expanded as this would generate an infinite recursion; they do however have many uses, *e.g.* as special markers and delimiters within code.

l3seq This implements data-types such as queues and stacks.

l3skip Implements the "rubber length" datatype `skip`, the "rigid length" datatype `dim`, and the math mode "rubber length" datatype `muskip`.

l3tl This implements a basic data-type, called a *token-list variable* (`tl var.`), used for storing named token lists: these are TeX macros with no arguments.

l3token Analysing token lists and token streams, including peeking ahead to see what's coming next and inspecting tokens to detect which kind they are.

6 Moving from LaTeX 2$_\varepsilon$ to LaTeX3

To help programmers to use LaTeX3 code in existing LaTeX 2$_\varepsilon$ package, some short notes on making the change are probably desirable. Suggestions for inclusion here are welcome! Some of the following is concerned with code, and some with coding style.

- expl3 is mainly focused on programming. This means that some areas still require the use of LaTeX 2$_\varepsilon$ internal macros. For example, you may well need `\@ifpackageloaded`, as there is currently no native LaTeX3 package loading module.
- User level macros should be generated using the mechanism available in the xparse package, which is part of the `l3package` bundle, available from CTAN or the LaTeX3 SVN repository.
- At an internal level, most functions should be generated `\long` (using `\cs_new:Npn`) rather than "short" (using `\cs_new_nopar:Npn`).
- Where possible, declare all variables and functions (using `\cs_new:Npn`, `\tl_new:N`, etc.) before use.
- Prefer "higher-level" functions over "lower-level", where possible. So for example use `\cs_if_exist:N(TF)` and not `\if_cs_exist:N`.
- Use space to make code readable. In general, we recommend a layout such as:

  ```
  \cs_new:Npn \foo_bar:Nn #1#2
    {
      \cs_if_exist:NTF #1
        { \__foo_bar:n {#2} }
        { \__foo_bar:nn {#2} { literal } }
    }
  ```

  where spaces are used around `{` and `}` except for isolated `#1`, `#2`, *etc.*
- Put different code items on separate lines: readability is much more useful than compactness.
- Use long, descriptive names for functions and variables, and for auxiliary functions use the parent function name plus `aux`, `auxi`, `auxii` and so on.
- If in doubt, ask the team via the LaTeX-L list: someone will soon get back to you!

7 Load-time options for expl3

To support code authors, the expl3 package for LaTeX 2$_\varepsilon$ includes a small number of load-time options. These all work in a key–value sense, recognising the `true` and `false` values. Giving the option name alone is equivalent to using the option with the `true` value.

`check-declarations` All variables used in LaTeX3 code should be declared. This is enforced by TeX for variable types based on TeX registers, but not for those which are constructed using macros as the underlying storage system. The `check-declarations` option enables checking for all variable assignments, issuing an error if any variables are assigned without being initialised. See also the function `\debug_check_declarations_on:`.

`log-functions` The `log-functions` option is used to enable recording of every new function name in the `.log` file. This is useful for debugging purposes, as it means that there is a complete list of all functions created by each module loaded (with the exceptions of a very small number required by the bootstrap code for LaTeX3). See also the function `\debug_log_functions_on:`.

`enable-debug` To allow more localized checking and logging than provided by `check-declarations` and `log-functions`, expl3 provides a few `\debug_...` functions (described elsewhere) that turn on the corresponding checks within a group. These functions can only be used if expl3 is loaded with the `enable-debug` option.

`driver` Selects the driver to be used for color, graphics and related operations that are driver-dependent. Options available are

`auto`	Let LaTeX3 determine the correct driver. With DVI output, this selects the `dvips` back-end for pdfTeX and LuaTeX, and `dvipdfmx` for pTeX and upTeX. This is the standard setting.
`latex2e`	Use the graphics package to select the driver, rather than LaTeX3 code.
`dvips`	Use the `dvips` driver.
`dvipdfmx`	Use the `dvipdfmx` driver.
`dvisvgm`	Use the `dvisvgm` driver.
`pdfmode`	Use the `pdfmode` driver (direct PDF output from pdfTeX or LuaTeX).
`xdvipdfmx`	Use the `xdvipdfmx` driver (XeTeX only).

8 Using expl3 with formats other than LaTeX 2$_\varepsilon$

As well as the LaTeX 2$_\varepsilon$ package expl3, there is also a "generic" loader for the code, `expl3.tex`. This may be loaded using the plain TeX syntax

```
\input expl3-generic %
```

This enables the programming layer to work with the other formats. As no options are available loading in this way, the "native" drivers are automatically used. If this "generic" loader is used with LaTeX 2$_\varepsilon$ the code automatically switches to the appropriate package route.

After loading the programming layer using the generic interface, the commands `\ExplSyntaxOn` and `\ExplSyntaxOff` and the code-level functions and variables detailed in interface3 are available. Note that other LaTeX 2$_\varepsilon$ packages *using* expl3 are not loadable: package loading is dependent on the LaTeX 2$_\varepsilon$ package-management mechanism.

9 Engine/primitive requirements

To use expl3 and the higher level packages provided by the team, the minimal set of primitive requirements is currently

- All of those from TeX90.
- All of those from ε-TeX *excluding* `\TeXXeTstate`, `\beginL`, `\beginR`, `\endL` and `\endR` (*i.e.* excluding TeX--XeT).
- Functionality equivalent to the pdfTeX primitive `\pdfstrcmp`.

Any engine which defines `\pdfoutput` (*i.e.* allows direct production of a PDF file without a DVI intermediate) must also provide `\pdfcolorstack`, `\pdfliteral`, `\pdfmatrix`, `\pdfrestore` and `\pdfsave` or equivalent functionality.

Practically, these requirements are met by the engines

- pdfTeX v1.40 or later.
- XeTeX v0.9994 or later.
- LuaTeX v0.70 or later.
- e-(u)pTeX mid-2012 or later.

Additional modules beyond the core of expl3 may require additional primitives. In particular, third-party authors may significantly extend the primitive coverage requirements.

10 The LaTeX3 Project

Development of LaTeX3 is carried out by The LaTeX3 Project. Over time, the membership of this team has naturally varied. Currently, the members are

- Johannes Braams
- David Carlisle
- Robin Fairbairns
- Morten Høgholm
- Bruno Le Floch
- Thomas Lotze
- Frank Mittelbach
- Will Robertson
- Chris Rowley
- Rainer Schöpf
- Joseph Wright

while former members are

- Michael Downes
- Denys Duchier
- Alan Jeffrey
- Martin Schröder

References

[1] Donald E Knuth *The TEXbook.* Addison-Wesley, Reading, Massachusetts, 1984.

[2] Goossens, Mittelbach and Samarin. *The LATEX Companion.* Addison-Wesley, Reading, Massachusetts, 1994.

[3] Leslie Lamport. *LATEX: A Document Preparation System.* Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[4] Frank Mittelbach and Chris Rowley. "The LATEX3 Project". *TUGboat*, Vol. 18, No. 3, pp. 195–198, 1997.