The expl3 package and philosophy*

The LaTeX3 Project†

2009/06/01

Abstract

This paper gives a brief 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.
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.
An earlier description of the LaTeX3 programming language is available from http://www.latex-project.org/papers.

1 Introduction

This paper describes the conventions for a TeX-based programming language which is intended to provide a more consistent and rational environment for the construction of large scale systems, such as LaTeX, using TeX macros.

Variants of this language have been in use by The LaTeX3 Project Team since around 1990 but the syntax specification to be outlined here should *not* be considered final. This is an experimental language thus many aspects, such as the syntax conventions and naming schemes, may (and probably will) change as more experience is gained with using the language in practice.

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.

*This file has version number 1381, last revised 2009/06/01.

†Frank Mittelbach, Denys Duchier, Chris Rowley, Rainer Schöpf, Johannes Braams, Michael Downes, David Carlisle, Alan Jeffrey, Morten Høgholm, Thomas Lotze, Javier Bezos, Will Robertson, Joseph Wright

The practical ideas behind this argument syntax will be explained, together with the semantics of the expansion control mechanism and the interface used to define variant forms of functions. The paper also discusses some advantages of the syntax for parameter names.

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 (especially when using a reasonably intelligent editor).

The final section gives some details of our plans to distribute parts of this system. Although the paper is now a little dated, more general information concerning the work of the LaTeX3 Project can be found in Ref. [4].

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 2$_\varepsilon$, as described in [3] and [2], or a mark-up language defined via SGML 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 SGML-based systems the DSSSL language may come to play this role. For LaTeX, this level was almost completely missing from LaTeX2.09; LaTeX 2$_\varepsilon$ 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.

Only the last of these three interfaces is covered by this paper, which describes a system aimed at providing a suitable basis for coding large scale projects in TEX (but this should not preclude its use for smaller projects). 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

The naming conventions for this programming language distinguish between *functions* and *parameters*. Functions can have arguments and they are executed. Parameters can be assigned values and they are used in arguments to functions; they are not directly executed but are manipulated by mutator and accessor functions. Functions and parameters with a related functionality (for example accessing counters, or manipulating token-lists, etc.) are collected together into a *module*.

Note that all these terms are only LATEX terminology and are not, for example, intended to indicate that the commands have these properties when considered in the context of basic TEX or in any more general programming context.

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 parameter names.

`\l_tmpa_box` is a local parameter (hence the `l_` prefix) corresponding to a box register.
`\g_tmpa_int` is a global parameter (hence the `g_` prefix) corresponding to an integer register (i.e., a TEX count register).
`\c_empty_toks` is the constant (`c_`) token register parameter that is for ever 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 syntax of the conventions

We shall now look in more detail at the syntax of these names.
The syntax of parameter names is as follows:

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

The syntax of function names is as follows:

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

3.3 Modules and descriptions

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 parameters. 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.

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.

As a semi-formalized concept the letter `g` is sometimes used to prefix certain parts of the ⟨*description*⟩ to mark the function as "globally acting", e.g., `\int_set:Nn` is a local operation while `\int_gset:Nn` is a global operation. This of course goes hand in hand with when to use `\l_` and `\g_` variable prefixes.

3.4 Parameters: access and type

The ⟨*access*⟩ part of the name describes how the parameter can be accessed. Parameters are primarily classified as local, global or constant (there are further, more technical, classes). This *access* type appears as a code at the beginning of the name; the codes used include:

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

g parameters whose value should only be set globally;

l parameters whose value should only be set locally.

The ⟨*type*⟩ will normally (except when introducing a new data-type) be in the list of available *data-types*; these include the primitive TEX data-types, such as the various registers, but to these will be added data-types built within the LATEX programming system.

Here are some typical data-type names:

int integer-valued count register;

tl token list variables: placeholders for token lists;

toks token register;

box box register;

skip 'rubber' lengths;

dim 'rigid' lengths;

num A 'fake' integer type using only macros. Useful for setting up allocation routines;

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

prop property list;

clist comma separated list;

stream an input or output stream (for reading from or writing to, respectively);

bool either true or false.

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

3.5 Functions: basic 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 characters, each denoting one argument of the function. It is important to understand that 'argument' here refers to the effective argument of the LATEX function, not to an argument at the TEX-level. Indeed, the top level TEX macro that has this name typically has no arguments. This is an extension of the existing LATEX convention where one says that `\section` has an optional argument and a mandatory argument, whereas the TEX macro `\section` actually has zero parameters at the TEX level, it merely calls an internal LATEX command which in turn calls others that look ahead for star forms and optional arguments.

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 and other commands that are provided for use only while bootstrapping the LATEX kernel. If the TEX primitive needs to be used in other contexts it will be given an alternative, more appropriate, name with a useful argument specification. The argument syntax of these is often weird, in the sense described next.

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. For example, `\seq_gpush:Nn` is a function that takes two arguments, the first is a single token (the sequence) and the second may consist of several tokens surrounded by braces.

This concept of argument specification makes it easy to read the code and should be followed when defining new functions.

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_io_file_name_seq` to keep track of them (`io` is the input-output file module). The basic operation here is to push a name onto this stack which could be done by the operation

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

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

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

```
\seq_gpush:Nn
   \g_io_file_name_seq
   \l_io_curr_file_tl
```

we will not get the value of the register pushed onto the stack, only the register name itself. Instead a suitable number of `\exp_after:wN` would be necessary (together with extra braces) to change the order of execution, i.e.

```
\exp_after:wN
   \seq_gpush:Nn
\exp_after:wN
      \g_io_file_name_seq
\exp_after:wN
      {\l_io_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 argument part of the function to mark the desired behavior. In the above example one would write

```
\seq_gpush:NV
   \g_io_file_name_seq
   \l_io_curr_file_tl
```

to achieve the desired effect. Here the `V` is for "retrieve the value of the register" (the second argument) 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_stack} \l_tmpa_tl
```

is equivalent to

```
\seq_gpush:NV \g_file_name_stack \l_tmpa_tl.
```

V Value of a register.
This means that the register in question is returned, be it an integer, a length type register, a macro storage register or similar. The value is returned to the function as a braced token list.

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 returns the 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. This means that expansion takes place until only unexpandable tokens are left.

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.

f 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 will gobble it and not expand the next argument.

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 \string #1}
               {?\string #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.

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@` according to our conventions. A function such as this but taking two normal '`n`' arguments might reasonably be named `\seq_test_in:Nn`; thus the variant function we need will be defined with the appropriate argument types and its name will be `\seq_test_in:cV`. Now this code fragment will be simply:

```
\seq_test_in:cV {sym#3} \l_group_seq
```

Note that, in addition to the lack of `\expandafter`, the space after the `}` will be 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 will provide variants with a range of argument forms, and similarly it is expected that extension packages providing new functions will make them available in the 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 will be identical and thus no conflict will occur.

5 Parameter assignments and accessor functions

5.1 Checking assignments

One of the advantages of having a consistent scheme is that the system can provide more extensive error-checking and debugging facilities. For example, an accessor function that makes a *global* assignment of a value to a parameter can check that it is not passed the name of a *local* parameter as that argument: it does this by checking that the name starts with `\g_`.

Such checking is probably too slow for normal use, but the code can have hooks built in that allow a format to be made in which all functions perform this kind of check.

A typical section of the source[1] for such code might look like this (recall that all white space is ignored):

[1] This code uses the docstrip system described in [2], Section 14.3.

```
%<*!check>
\cs_new_eq:NN
  \toks_gset:Nn \tex_global:D
%</!check>
%<*check>
\cs_new_nopar:Npn
  \toks_gset:Nn #1
    {
    \chk_global:N #1
    \tex_global:D #1
    }
%</check>
```

In the above code the function `\toks_gset:Nn` takes a single token (`N`) specifying a token register, and globally sets it to the value passed in the second argument.

A typical use of it would be:

```
\toks_gset:Nn \g_xxx_toks {<some value>}
```

In the normal definition, `\toks_gset:Nn` can be simply `\let` to `\global` because the primitive TEX token register does not require any explicit assignment function: this is done by the `%<*!check>` code above.

The alternative definition first checks that the argument passed as `#1` is the name of a global parameter and raises an error if it is not. It does this by taking apart the command name passed as `#1` and checking that it starts `\g_`.

5.2 Consistency

The primitive TEX syntax for register assignments has a very minimal syntax and, apart from box functions, there are no explicit functions for assigning values to these registers.

This makes it impossible to implement alternative data-types with a syntax that is both consistent and at all similar to the syntax for the primitives; moreover, it encourages a coding style that is very error prone.

As in the `\toks_gset:Nn` example given above, all LATEX data-types are provided with explicit functions for assignment and for use, even when these have essentially empty definitions. This allows for better error-checking as described above; it also allows the construction of further data-types with a similar interface, even when the implementation of the associated functions is very complex.

For example, the 'fake-integer' (`num`) data-type mentioned above will appear at the LATEX programming level to be exactly like the data-type based on primitive count registers; however, internally it makes no use of count registers. Typical functions in this module are illustrated here.

```
\num_new:N \l_tmpa_num
```

This declares the local parameter `\l_example_num` as a fake-counter.

```
\num_add:Nn \l_example_num \c_thirty_two
```

This increments the value of this fake-counter by 32.

6 The experimental distribution

The initial implementations of a LaTeX programming language using this kind of syntax remain unreleased (and not completely functional); they partly pre-date LaTeX 2_ε! The planned distribution will provide a subset of the functionality of those implementations, in the form of packages to be used on top of LaTeX 2_ε.

The intention is to allow experienced TeX programmers to experiment with the system and to comment on the interface. This means that ***the interface will change***. No part of this system, including the name of anything, should be relied upon as being available in a later release. Please do *experiment* with these packages, but do *not* use them for code that you expect to keep unchanged over a long period.

In view of the intended experimental use for this distribution we shall, in the first instance, produce only a few modules for use with LaTeX 2_ε. These will set up the conventions and the basic functionality of, for example, the expansion mechanism; they will also implement some of the basic programming constructs, such as token-lists and sequences. They are intended only to give a flavour of the code: the full LaTeX3 kernel will provide a very rich set of programming constructs so that packages can efficiently share code, in contrast with the situation in the current LaTeX where every large package must implement its own version of queues, stacks, etc., as necessary.

At time of writing the expl3 bundle consists of the following modules.

This distribution will also contain the LaTeX source for the latest version of this document, a docstrip install file and three small test files.

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

l3box Primitives for dealing with boxes.

l3calc A re-implementation of the LaTeX 2_ε package calc package in expl3 that provides extended methods for numeric and dimensional calculations and assignments.

l3chk A module that provides functionality comparable to LaTeX's `\newcommand` and `\renewcommand`, and also the extra level of checking for ensure internal consistency in the code.

l3clist Methods for manipulating comma-separated token lists.

l3expan This is the argument expansion module discussed above.

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

l3io A module providing low level input and output functions.

l3keyval A re-implementation of the LaTeX 2_ε package keyval; provides low-level macros for dealing with lists of the form `{ key1=val1 , key2=val2 }`.

l3messages/l3msg Communicating with the user.

l3names This sets up the basic naming scheme and renames all the TeX primitives. If it is loaded with the option `[removeoldnames]` then the old primitive names such as `\box` become *undefined* and are thus available for user definition. Caution: use of this option will certainly break existing TeX code!

l3num This implements the 'fake integer' datatype `num`.

l3precom A 'pre-compilation' module that provides functions dealing with pointer creation and handling, and using external files to record the state of the current definitions.

l3prg Program control structures such as boolean data type `bool`, generic do-while loops, case-switches, sorting routines and stepwise loops.

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 will generate 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` and the 'rigid length' datatype `dim`.

l3tl This implements a basic data-type, called a *token-list variable* (`tl var.`), used for storing named token lists: these are essentially 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.

l3toks A data-type corresponding to TeX's primitive token registers.

l3xref Data structure for low-level document cross-referencing. This provided the foundation for Heiko Oberdiek's zref package.

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.

6.1 expl3 implementation

Well, it's not complicated :)

<*package>

```
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
```

We already loaded l3names at the beginning of the DTX file.

The l3chk package is omitted since it is only used for conditional processing with full error-checking turned on. Most users will generally not need to do this, and we haven't set it up yet, anyway.

Fundamentals:

```
\RequirePackage{
  l3basics,
  l3expan,
  l3tl,
  l3num,
  l3intexpr,
  l3quark,
  l3seq,
  l3toks,
  l3int,
  l3prg,
  l3clist,
  l3token,
  l3io,
  l3prop,
  l3msg,
  l3skip,
}
```

All the rest:

```
\RequirePackage{
  l3box,
  l3keyval,
  l3precom,
  l3calc,
  l3xref,
  l3file
}
```

</package>