Programming key–value in expl3

Joseph Wright

1 Introduction

Key–value entry, in which a series of ⟨*key*⟩ = ⟨*value*⟩ statements are given in a comma-separated list, is a powerful method for specifying a flexible range of values in (LA)TEX. For the user, this type of input avoids the need for a very large number of control macros to alter how a LATEX package or class behaves. Using key–value input also allows the programmer to expose a very wide range of internal settings. Used properly, this should avoid the need for users to access or modify the internal code of a LATEX package or class to achieve the desired behaviour.

Programming key–value methods can be confusing, as the link between the user interface and implementation at the code level is not always obvious. Christian Feuersänger and I have given an overview of programming key–value input using most of the LATEX 2_ε implementations (Wright and Feuersänger, 2010), covering both the broad concepts and the implementation details.

The new possibilities opened up by the expl3 programming language of LATEX3 (LATEX3 Project, 2010) include providing a new, consistent interface for creating key–value input. Programming using expl3 is somewhat different from traditional (LA)TEX, and I covered the key general concepts for using expl3 in an earlier *TUGboat* article (Wright, 2010). In this article, I will focus on how expl3 implements key–value methods, with an assumption of some familiarity with both expl3 and using key–value with LATEX 2_ε.

2 Low-level key–value support: l3keyval

The expl3 bundle of modules includes two with key in their name, l3keyval and l3keys. The two are aimed at different parts of the programming process, and most of this article will focus on l3keys. However, some idea of the place of l3keyval in the larger scheme is useful.

The l3keyval module provides low-level parsing of key–value input, used by l3keys but also available for other purposes. Thus, l3keyval does not do anything beyond split a list first at each comma (that is, into key–value pairs), and then into a key and value. It includes the facility to sanitize the category codes of '`,`' and '=', and also to ignore spaces if necessary.

The result is that, while l3keyval is crucial for key–value support using expl3, the exact mechanisms used are unimportant. This frees us to focus on the facilities provided by l3keys. (Well, it frees *you* from understanding l3keyval: I wrote most of l3keys, so I have to know what is going on!)

3 The design ideas for l3keys

Perhaps the best package for defining key–value input using LATEX 2_ε methods is pgfkeys (Tantau, 2008). It uses key–value input in the definition of keys themselves, and thus uses the power of key–value methods to help the programmers as well as the end user. The approach taken by the l3keys module in expl3 is inspired by pgfkeys, although programmers familiar with the latter will find some important differences.

The main design ideas for l3keys were:

- Use of key–value methods for creating keys at the programming level;
- Separate functions for defining and setting keys (namely, `\keys_define:nn` and `\keys_set:nn`);
- 'Fit' with the LATEX3 syntax and variable conventions;
- Rich set of key types, including clear handling of multiple choices.

Taking these ideas and concepts from pgfkeys, l3keys introduces the idea of 'properties' for keys. Each valid key name must have at least one property, to attach some code to the key. By combining a number of properties, a wide range of effects can be created without an overly-complex interface.

4 Functions for keys

Keys are created using the `\keys_define:nn` function, where the function name follows general expl3 conventions and thus requires two arguments. The first is the module name with which the keys are associated. Typically, this will be the same as the LATEX 2_ε package or LATEX3 module name being created, although it can be more complex. The second argument for `\keys_define:nn` is a list of keys, properties and values, which are then used to set up the key–value system.

```
\keys_define:nn { module } {
  key-one .property-a:n = value-one   ,
  key-one .property-b:n = value-two   ,
  key-two .property-a:n = value-three ,
}
```

As illustrated, the 'properties' of a key are indicated starting with a full stop (period) character at the end of the name of the key. (Remember that expl3 code blocks ignore spaces, so there are no significant spaces in the example.) In line with expl3 conventions, each property includes a specification to indicate what arguments it expects.

The second part of using key–value methods is setting keys, and is handled by the `\keys_set:nn` function. This also takes the module as the first argument and a key–value list as the second:

```
\keys_set:nn { module } {
  key-one = value-one   ,
  key-one = value-two   ,
  key-two = value-three ,
}
```

Here, the key–value list is used along with the key implementation to 'set' the keys.

`\keys_define:nn` will almost always appear inside code blocks, and so does not carry out any sanity checks on category codes of its input. On the other hand, `\keys_set:nn` is likely to handle user input, and so does carry out these checks. It is also worth saying that `\keys_set:nn` will typically be 'wrapped up' in a user-accessible function, say, `\modulesetup`. Using the LaTeX3 xparse module, this might look like:

```
\DeclareDocumentCommand
  \modulesetup { +m } {
    \keys_set:nn { module } {#1}
  }
```

or using traditional LaTeX 2ε:

```
\newcommand \modulesetup [1] {
  \keys_set:nn { module } {#1}
}
```

5 Key properties

The most general property that can be given for a key is `.code:n`. This associates completely general code with a particular key name; the value given to a key when used is available within the code as `#1`.

```
\keys_define:nn { module } {
  key .code:n = You~gave~input~#1! ,
  ...
}
```

As is generally the case with key–value input, we do not need braces around the code here, as it is delimited by the comma separating key–value pairs. The only exception is if the code itself contains `,` or `=` characters, which of course need to be 'hidden'.

Related to the `.code:n` property is `.code:x`. Following expl3 conventions, the difference here is in the expansion of the code. `.code:n` carries out no expansion, whereas `.code:x` carries out an `\edef`-like procedure:

```
\tl_set:Nn \l_tmp_tl { You~said }
\keys_define:nn { module } {
  key-one .code:x = \l_tmp_tl\~``#1'',
  key-two .code:n = \l_tmp_tl\~``#1'',
}
\tl_set:Nn \l_tmp_tl { You~typed }
```

If inside the document body we then do

```
\keys_set:nn { module } {
  key-one = text      ,
  key-two = more~text ,
}
```

the result will be to print 'You said "text"' followed by 'You typed "more text"'.

5.1 Storing values

One of the most common tasks to carry out using key–value methods is storing values in variables. While this can be done using the `.code:n` property, a series of dedicated properties are available, all of which follow the same general pattern.

```
\keys_define:nn { module } {
  key-one   .dim_set:N  = \l_module_dim  ,
  key-two   .int_set:N  = \l_module_int  ,
  key-three .skip_set:N = \l_module_skip ,
  key-four  .tl_set:N   = \l_module_tl   ,
}
```

As illustrated, each property should 'point' to a variable to store the value given. While the examples here use LaTeX3-style variables, the properties will also work with variables following LaTeX 2ε naming conventions. Giving the setting instruction:

```
\keys_set:nn { module } {
  key-four = content
}
```

will set token list variable `\l_module_tl` to the text `content`. (As a reminder, a LaTeX3 'token list variable' is a macro which is used as a variable to store tokens, often text.)

Assignments using the `.`⟨*var*⟩`_set:N` properties are local, which is normally what you want. However, global assignments can also be made, using the `.`⟨*var*⟩`_gset:N` properties. These are set up and used in exactly the same way as the local versions:

```
\keys_define:nn { module } {
  key-one .tl_set:N  = \l_module_tl ,
  key-two .tl_gset:N = \g_module_tl ,
}
\keys_set:nn { module } {
  key-one = text, % Locally
  key-two = text  % Globally
}
```

Values can be stored in token list variables either as-given or with full (`\edef`) expansion. As the expansion takes place later it cannot be indicated using an argument specifier. Instead, two 'expand then store' properties are available:

```
\keys_define:nn { module } {
  key-one .tl_set_x:N  = \l_module_tl ,
  key-two .tl_gset_x:N = \g_module_tl ,
}
\tl_set:Nn \l_tmp_tl { text }
\keys_set:nn { module } {
  key-one = \l_tmp_tl ,
  key-two = \l_tmp_tl ,
}
\tl_set:Nn \l_tmp_tl { changed }
```

Both `\l_module_tl` and `\g_module_tl` will store '`text`', whereas with the normal `.tl_(g)set:N` property they would simply contain '`\l_tmp_tl`'.

One thing to notice is that l3keys will make sure every variable we use actually exists. So there is no need to worry about long lists of declarations along with an equally long list setting up keys.

5.2 Storing Boolean values

Boolean variables can only take true and false values, and so can be viewed as a type of multiple choice. To avoid code duplication, l3keys provides a method to set LaTeX3 Boolean variables using a pre-defined choice, using the `.bool_set:N` property. This works in much the same way as those for setting other variables, except that it will only accept the values `true` and `false`.

```
\keys_define:nn { module } {
  key .bool_set:N = \l_module_bool
}
```

It is important to note that LaTeX3 Boolean variables do *not* work in the same way as TeX or LaTeX 2$_\varepsilon$ `\if..` statements. Thus, `.bool_set` cannot be used to set the latter: you have to use `.code:n`.

5.3 Values assumed, required, forbidden

Some keys can assume a particular value is meant if only the key name is given. This is often the case with keys which can be set only to `true` or `false`: giving the key name alone is usually the same as given the `true` value. This is referred to by l3keys as a default value, and is set up using the `.default:n` property:

```
\keys_define:nn { module } {
  key .code:n    = Do stuff with #1! ,
  key .default:n = yes
}
```

With the settings above

```
\keys_set:nn { module } { key }
```

and

```
\keys_set:nn { module } { key = yes }
```

are entirely equivalent.

Alternatively, rather than assume a particular value is meant if the key name alone is given, you might wish to always require a value or forbid one entirely. This can be controlled using the `.value_required:` and `.value_forbidden:` properties, both of which act in an obvious way:

```
\keys_define:nn { module } {
  key-one .code:n = Do stuff with #1,
  key-one .value_required:          ,
  key-two .code:n = Do other stuff  ,
  key-two .value_forbidden:         ,
}
```

In both cases, error messages result if the requirement is not met.

5.4 Choices

One very useful thing to do using key–value input is to provide a list of predetermined choices. These can then be used to set up potentially complicated code patterns with a simple interface.

A key is made into a multiple choice by setting the `.choice:` property, but this does not create any valid choices! Each choice is created as a 'subkey':

```
\keys_define:nn { module } {
  key .choice:,
  key / choice-a .code:n = Choice-a code ,
  key / choice-b .code:n = Choice-b code ,
  key / choice-c .code:n = Choice-c code ,
}
```

As shown, each choice is given in the format ⟨*key*⟩ `/` ⟨*choice*⟩: the `/` character marks the boundary between the key and subkey names. It is likely that there will be some similarity between the implementation for different keys, but this is not necessary for the system to work.

To avoid the need to duplicate code between choices with very similar implementations, an automated system is available. First, the shared code is set up using the `.choice_code:n` property. A comma-separated list of choices is then given using the `.generate_choices:n` property.

```
\keys_define:nn { module } {
  key .choice_code:n = {
    Do something using either
    \l_keys_choice_tl or
    \l_keys_choice_int.
  },
  key .generate_choices:n = {
    choice-a, % Choice 0
    choice-b, % Choice 1
    choice-c, % Choice 2
    ...
  }
}
```

As illustrated, within the code `\l_keys_choice_tl` and `\l_keys_choice_int` are available. The name of the current choice (for example `choice-b`) is assigned to `\l_keys_choice_tl`, its numeric position in the list (for example `1` for `choice-b`) is assigned to `\l_keys_choice_int`. Notice that this is indexed from 0!

5.5 Keys setting keys

The final property provide by l3keys is for creating so-called meta keys: keys which themselves set other keys. Using the `.meta:n` property, we can provide a short-cut to set several things in one go.

```
\keys_define:nn { module } {
  key-one   .code:n = Some code  ,
  key-two   .code:n = Other code ,
  key-three .meta:n = {
    key-one = Value ,
    key-two = Value ,
  }
}
```

It is possible to pass on the argument given to a meta-key to its 'children' using `#1`:

```
\keys_define:nn { module } {
  key-one   .code:n = Something with #1 ,
  key-two   .code:n = Other thing #1    ,
  key-three .meta:n = {
    key-one = #1 ,
    key-two = #1 ,
  }
}
```

Almost always, the data for a meta key needs to be wrapped in braces, as it contains `,` and `=` characters.

6 Unknown keys and choices

The ability to handle input which has not been previously defined is important for flexible key–value methods. Each time a key is set using `\keys_set:nn`, after looking for the key itself l3keys checks for a special `unknown` key before issuing an error message. This key is set up in the same way as any other, and can carry out whatever function is appropriate. The name of the unknown key is available within the `unknown` key as `\l_keys_key_tl`, and can therefore be used by the attached code. A simple example would be to issue a customised error message if a key is unknown:

```
\keys_define:nn { module } {
  unknown .code:n = {
    \msg_error:nnx { module }
      { unknown-key } { \l_keys_key_tl }
  }
}
```

More sophisticated use might include creating new keys from this data, storing information in custom variables and so on.

7 LaTeX 2_ε package options

As LaTeX3 development is still at the stage of creating low-level structures, the most likely use of l3keys is with LaTeX 2_ε packages and classes. To enable the methods described here to be used with LaTeX 2_ε package and class options, a support package l3keys2e is available to enable the appropriate processing.

As is true with any key–value package, any options created with l3keys are simply keys that have been defined when option processing takes place. So creating options means first using `\keys_define:nn` for set up, then processing the option list with the `\ProcessKeysOptions` function. This takes a single argument: the name of the module.

```
\keys_defined:nn { module } {
  option-one .code:n = ... ,
  option-two .code:n = ... ,
  ...
}
\ProcessKeysOptions { module }
```

8 An example

Putting everything together can be challenging starting from a bare description of the methods available. In my general key–value article, I included a short example package to illustrate some of the major ideas. I'll use the same scenario here, which also means that readers can compare the l3keys approach directly to keyval- and pgfkeys-based solutions.

Consider the following situation. The inexperienced LaTeX user who asked for a small package for the last article has come back, and wants to be at the cutting edge. So they've asked if you can rewrite your code from before using expl3. What they want is a package to provide one user macro, `\xmph`, which will act as an enhanced version of `\emph`. As well as italic, it should be able to make its argument bold, coloured or a combination. This should be controllable on loading the package, or during the document. Finally, a de-activation setting is requested, so that the `\xmph` macro acts exactly like `\emph`. This latter setting should be available only in the preamble, so that it will apply to the entire document body.

Looking back over your earlier solution, there is not too much to change. You decide to follow LaTeX3 conventions and adjust some of the option names slightly:

- `inactive`, a key with no value, which can be given only in the preamble;

- `use-italic`, a Boolean option for making the text italic;
- `use-bold` and `use-colour`, two more Boolean options with obvious meanings;
- `colour`, a string option to set the colour to use when the `use-colour` option is true.

You also anticipate that US users would prefer the option names `use-color` and `color`, and so you decide to implement them as well.

Things are going to look a bit different from a traditional LaTeX 2_ε package, but hopefully things will not be too bad! The first stage is to declare the code as a LaTeX3 package, and to load color for colour support, l3keys2e to do the option processing, and xparse to make user commands the LaTeX3 way.

```
\RequirePackage{color,l3keys2e,xparse}
\ProvidesExplPackage
  {xmph} {2010/01/02}
  {2.0}  {Extended emph}
```

The next stage is to set up the key–value input, and set the default values (red italic text).

```
\keys_define:nn { xmph } {
  colour
    .tl_set:N   = \l_xmph_colour_tl   ,
  color
    .meta:n     = { colour = #1 }     ,
  inactive
    .code:n     =
      \cs_set_eq:NN \xmph \emph       ,
  use-bold
    .bool_set:N = \l_xmph_bold_bool   ,
  use-colour
    .bool_set:N = \l_xmph_colour_bool ,
  use-color
    .bool_set:N = \l_xmph_colour_bool ,
  use-italic
    .bool_set:N = \l_xmph_italic_bool ,
}
\keys_set:nn { xmph } {
  colour = red ,
  use-italic
}
```

With everything set up, any load-time options can be dealt with using the `\ProcessKeysOptions` function.

```
\ProcessKeysOptions { xmph }
```

For the code implementing everything, the pattern here is the same as in the LaTeX 2_ε version. The formatting functions are wrapped up one inside another.

```
\NewDocumentCommand \xmph { m } {
  \xmph_emph:n {
    \xmph_bold:n {
      \xmph_colour:n {#1}}}}
\cs_new:Nn \xmph_bold:n {
  \bool_if:NTF \l_xmph_bold_bool {
    \textbf {#1}
  }{#1}}
\cs_new:Nn \xmph_colour:n {
  \bool_if:NTF \l_xmph_colour_bool {
    \textcolor { \l_xmph_colour_tl } {#1}
  }{#1}}
\cs_new:Nn \xmph_emph:n {
  \bool_if:NTF \l_xmph_italic_bool {
    \emph {#1}
  }{#1}}
```

The last job to do is to disable the `inactive` at the end of the preamble. That simply means setting the option to do nothing.

```
\AtBeginDocument {
  \keys_define:nn { xmph } {
    inactive .code:n = { }
  }
}
```

9 Conclusions

Key–value methods are a powerful way to provide users with a clear interface to code internals. expl3 adds the ability to create key–value input to LaTeX, along with the many other programming refinements it provides. By including this in the base layer of LaTeX3, the confusion between LaTeX 2_ε implementations is avoided. This should mean that more people can get to grips with using key–value methods in their packages, and do so more reliably.

References

LaTeX3 Project. “The expl3 package”. Available from CTAN, `macros/latex/contrib/expl3`, 2010.

Tantau, Till. “pgfkeys”. Part of the Ti*k*Z and pgf bundle, available from CTAN, `graphics/pgf`, 2008.

Wright, Joseph. “LaTeX3 programming: External perspectives”. *TUGboat* **31**, 2010.

Wright, Joseph, and C. Feuersänger. “Implementing key–value input: an introduction”. *TUGboat* **31**, 2010.

◇ Joseph Wright
Morning Star
2, Dowthorpe End
Earls Barton
Northampton NN6 0NH
United Kingdom
`joseph dot wright (at)`
`morningstar2 dot co dot uk`