Key–value setting handling in the LaTeX kernel

Joseph Wright, LaTeX Project Team

1 Introduction

LaTeX 2$_\varepsilon$ introduced the idea of classes and packages, and along with that the concept of class and package options. These are nowadays very familiar to LaTeX authors, with the first optional argument to `\documentclass` and `\usepackage` used in the vast majority of LaTeX documents:

```
\documentclass[10pt,final]{article}
\usepackage[T1]{fontenc}
\usepackage[numbers]{natbib}
```

This system is a powerful way of controlling behaviours, but is limited as the options are string literals. This is perhaps best exemplified by the font size options: `10pt`, `11pt` and `12pt` in the standard classes. Rather than being dimensions parsed and interpreted, these options are used to load hard-coded configurations for the three nominal sizes.

For the programmer, creating options is easy, requiring only one command for each option, plus a separate one to process the list of options given by the user:

```
\DeclareOption{foo}{...}
\DeclareOption{bar}{...}
\ProcessOptions\relax
```

2 Third-party key–value support

It is natural to want to have a more flexible system, and key–value methods are the obvious way to obtain this. As the kernel hasn't to-date offered this, a number of third-party packages have been developed to support programmers in providing these interfaces.

Fundamentally, all of these packages work in the same way. Keys are defined using an existing key–value implementation, ready to be used when the options are examined. A dedicated command is provided to do the latter, and this examines each option recorded by LaTeX and tries to match it as a key or a key–value pair. If that is possible, the key–value process is called, while if the option is not known as a key, an unknown key process is used.

Perhaps the most convenient package for providing key–value options to date is kvoptions, written by Heiko Oberdiek. Rather than try to provide a full setup for generic key–value work, this package provides a small set of commands which define the most common types of key–value interface with clear names:

```
\RequirePackage{kvoptions}
\DeclareStringOption[me]{name}
\DeclareBoolOption{draft}
\DeclareComplementaryOption{final}{draft}
\DeclareDefaultOption{\ERROR}
\ProcessKeyvalOptions*
```

As we can see, kvoptions supports 'strings' (saving the tokens given in the input), boolean (switch) options and inverse booleans. It also provides a way to handle unknown options.

Using kvoptions or similar approaches has allowed programmers to provide key–value options for a number of years. But there are some downsides. First, ideally one wouldn't need to load a package to do this. It would also be better if there was one mechanism, not several with slightly different syntaxes. More fundamentally, the LaTeX kernel carries out space stripping and expansion of options *before* they are passed to packages to examine. This makes handling some option texts awkward.

There is also the issue of option clashes. The LaTeX kernel checks that the option list of a package is *identical* if you try to load it twice:

```
\usepackage[option = a]{mypkg}
\usepackage[option = b]{mypkg}
```

This is an issue even without key–value options, but with them it's much worse: it's not possible to know if there is a true clash or if the settings simply can override each other. So there needs to be a way to let packages themselves handle this.

3 The new kernel mechanism

In the 2022-06 release of the LaTeX kernel, a new built-in approach is available for processing options using key–value methods. This is based on the expl3 module l3keys, which is nowadays built into the kernel. You don't, though, need to know anything about expl3 to use the new approach: everything is made available under standard LaTeX 2$_\varepsilon$ names.

As for the classical approach, we need to do three things: create options (keys), define how to deal with unknown options, and process the options. Unlike `\DeclareOption`, the new command `\DeclareKeys` can create multiple options in one go. Each option (key) is created by setting one or more *properties*: these are given after the key name as `.`⟨*prop*⟩. The basic properties are `.store`, `.if` and `.code`. These set up keys, respectively, to store the input, to use it to set a switch or to insert arbitrary code. We can also set the `.default` for a key: the value that is assumed if none is given by the user. We will also be adding `.notif` for the Fall 2022 release: that will be the same as kvoptions' `\DeclareComplementaryOption`.

```
\DeclareKeys{
  name   .store   = \mypkg@name  ,
  name   .default = me           ,
  draft  .if      = mypkg@draft  ,
%  final .notif   = mypkg@draft  ,
  demo   .code    =
    \protected@edef\mypkg@demo{#1}
}
```

To deal with unknown keys, we can declare a dedicated handler: here we simply issue an error.

```
\DeclareUnknownKeyHandler{%
  \PackageError{mypkg}
    {Unknown option "\CurrentOption"}
    \@ehc
}
```

Finally we need to process the options: the command name here is pretty simple.

```
\ProcessKeyOptions
```

This approach will provide the key–value setup we want, and the kernel will automatically use the new approach to repeated loading: there will be no option clash warnings. We might, though, want more control: that can be obtained using the `.usage` property:

```
\DeclareKeys{
  name  .store   = \mypkg@name ,
  name  .default = me          ,
  name  .usage   = load        ,
  draft .if      = mypkg@draft ,
  draft .usage   = preamble    ,
  demo  .code    =
    \protected@edef\mypkg@demo{#1}
}
```

With this, the kernel will automatically issue an error if an option is used in the wrong place: after first loading for `load` options, outside of the preamble for `preamble` options.

4 Options are keys

You might have picked up from the above that 'options' and 'keys' are used almost interchangeably. That's because, when processing key–value options, they are simply keys that are created before a call to `\ProcessKeyOptions`. That means that we can use options as keys: all we need is a way to set them. This is available in a command called `\SetKeys`: you might notice the name is similar to the long-standing `\setkeys` from the keyval package.

As most of the time we want to set keys *after* loading a package, we need to pass the *family* the keys are in. This is given in an optional argument to `\SetKeys`. (The optional argument applies to all of the other new commands, but most of the time we don't need to worry about it, as LaTeX will automatically use the package or class name.) So we might have something like

```
\NewDocumentCommand\mypkgsetup{m}{%
  \SetKeys[mypkg]{#1}%
}
```

We can then use this new setup command to work with exactly the same keys as we have created as options: provided of course we do not try to do that outside their `usage` scope.

5 More flexibility

As the new approach is based on l3keys, we can use any key properties that are defined by l3keys. That is because of the fact that options are keys and *vice versa*: all we need to do is define the keys in the right place.

◇ Joseph Wright
Northampton, United Kingdom
`joseph dot wright (at) morningstar2.co.uk`

◇ LaTeX Project Team