# Keys and Values

## *new developments and mechanisms in key processing*

**Abstract**
This article introduces the xkeyval package as an extension of the well-known keyval package. The package provides more flexible commands, syntax enhancements, and a new option processing mechanism for class and package options using the *key=value* syntax.

**Keywords**
keyval, xkeyval, package options, pointers, presets, category codes, PSTricks

### Introduction

The keyval package [2] written by David Carlisle is widely used by package authors to provide the means for users to easily specify a lot of optional arguments for macros. The main advantages of using keyval are that the number of optional arguments is no longer limited to 9 and that the arguments are named and hence there is less chance of confusion about the syntax of a macro.

The package provides means to define key macros which handle the input of the user. These key macros have the form `\KV@family@keyname` and take one argument to handle user input. A key macro can, for instance, be defined by

```
\define@key{family}{pi}%
  {\setlength{\parindent}{#1}}
```

The key macros will be called at the use of `\setkeys` and this will set the keys. When `pi` is used, the key macro will set `\parindent`. Another typical example of its use is

```
\setkeys{family}{pi=10pt,pn=Page~\thepage}
```

The packages keyval and xkeyval are mainly directed to class and package authors. The `\define@key` command usually goes into the preamble or the package and the main interface for users is given by `\setkeys`.

### Why a new package?

When working on another package, the need arose to have multiple families in the package. Each family would provide keys for a particular macro or environment. This provided the means to block the use of illegal keys in a macro argument, which could have a destructive effect on the rest of the document. However, it would also be nice to be able to allow the user to set specific keys of each macro or environment globally in the preamble. One could, for instance, think of allowing the user to set the markup of all `example` and `exercise` environments in the document in the preamble, but disallowing changing the markup of `example` environments in `exercise` environments and vice versa. In more complicated settings, specifying keys in macros which are not designed to handle those keys can easily lead to almost untraceable errors. That was the start of the xkeyval package [1].

However, in the process of generalizing keyval, we noticed that a lot of packages had already tried extending the features, all in their own way. Quite some packages, for instance, include a system to allow the use of keys and values in `\usepackage` commands. Most famous examples are the hyperref, geometry and beamer packages. All of these approaches differ in details and are not portable to other packages without reprogramming. This called for a unified approach.

Another extra feature, found for instance in the hyperref package, is the availability of boolean keys which can only be true or false. hyperref actually implements this within the ordinary key system, using `\define@key`. However, since the function that needs to be executed on the use of the key is known (namely, set an “if” command to true or false depending on the input), the system can be simplified.

A final motivation for the new package is based on the fact that the development of the keyval package seemed to have paused since 1999 and that fundamental changes and improvements to the system could more easily be made with a new package. Among the improvements, we find the pointer syntax, the preset

system, the use of multiple families in \setkeys, robust input parsing and the support for the PSTricks family of packages. The remaining sections of this article will discuss these new developments.

## Keys and values in package options

First of all, the package supplies macros to declare class or package options, execute them and process them. The macros are available under the usual LaTeX names, but all with a postfix, namely

```
\DeclareOptionX
\ExecuteOptionsX
\ProcessOptionsX
```

These commands allow the user to assign a value to an option just as when using \setkeys. The first macro is based on \define@key and the final two are based on \setkeys. When mypack is using these commands, a user could for instance do

```
\usepackage[textcolor=red,font=times]{mypack}
```

These macros are fully integrated with the LaTeX option system. This, for instance, allows packages to copy global options specified in the \documentclass command, to pass options to other classes or packages and to update the list of unused global options that will be displayed by LaTeX in the log file.

However, key values like author=\textit{Me} in class or package options are not allowed, although they could easily be processed by \setkeys. This restriction results from the design of LaTeX's option processing mechanism which expands the entire option list (keys and values) completely, causing obvious trouble.[1]

To avoid these premature expansions, several kernel macros need to be redefined. xkeyval includes the xkvltxp package which contains these new definitions. Loading this package before loading the class or package which uses xkeyval for option processing, will allow class and package options to contain expandable macros. This file will not be included in the LaTeX $2_\varepsilon$ kernel since it might introduce compatibility conflicts for those using an old kernel but new packages which might depend on this new functionality.

## Prefixes, families and pointers

The package provides extended syntax for all of the commands provided by keyval.[2] The syntax for defining keys has been extended with an optional argument to set the prefix of the key macro. It is a good custom for package authors to use a package specific prefix for all internal macros as to avoid possibly redefining a macro of another package. Moreover, this optional argument allows for defining and setting keys in specialized systems such as implemented in the PSTricks package. More details about this system will be discussed in the section about the pst-xkey package.

The syntax for setting keys using \setkeys has been adjusted accordingly. Also, one can specify a list of families which should be scanned when setting keys, as discussed in the introduction. For instance,

```
\setkeys{font,page}{fs=10pt,pn=Page~\thepage}
```

Part of the new syntax is the possibility to use pointers to keys. Pointers allow to assign to keyb the value that has been assigned to keya, irrespective of what that value is. For example

```
\setkeys{family}{\savevalue{keya}=red,%
  keyb=\usevalue{keya}}
```

Here, \savevalue will make xkeyval save the value submitted to keya. \usevalue will use this value again. Note that one can use the \savekeys command to avoid typing \savevalue every time. If, in this example, red is changed to blue no changes are necessary to the value of keyb to assign it blue as well. This is an obvious similarity to TeX's behaviour in the macro case \def\cmdb{\cmda}.

This pointer system can be used as well in the default value system. This system submits a default value to the key macro in case the user has used the particular key, but didn't assign a value to it. One could, for example, define the keys

```
\define@key{fam}{keya}{keya: #1 }
\define@key{fam}{keyb}[\usevalue{keya}]{keyb: #1 }
```

Then the following use of \setkeys

```
\setkeys{fam}{\savevalue{keya}=test,keyb}
```

would result in typesetting

```
keya: test keyb: test
```

We will discuss some technical details regarding the pointer syntax. First of all, the control sequences \savevalue and \usevalue are not defined. Instead, the package considers these as delimiters. A simple parsing step will determine if \savevalue has been used in the key name part. Parsing is also used to substitute occurrences of \usevalue by the saved value. When a pointer is replaced, its replacement will also be scanned again for pointers. This allows for nested pointers in key values. Moreover, it makes sure that, once the value is submitted to a key macro, this value does not contain pointers anymore.[3]

The replacement process is a little bit more tricky when the user did not submit a value to the key. In this

case, the default value should be scanned for pointers. The default value macro for a key macro looks like `\prefix@fam@key@default` and has been defined to expand to

```
\prefix@fam@key{the default value}
```

This system has been introduced by keyval and a lot of packages use it. However, some packages do not use it in the way intended by keyval. For instance, the fancyvrb package defines default value macros to execute some code rather than to call the key macro and submit the default value to that. To retain compatibility with existing packages, it is impossible to change the setup of the default value system, only save the default value in the default value macro and submit this value to the key macro.

This is an important restriction for the pointer system since we want to retrieve the default value from the default value macro and scan it for pointers. The way the package proceeds is the following. It first checks whether the default key macro starts as expected, namely with the key macro name. If that is the case, it locally redefines the key macro to save the value to a macro and it executes the key macro. The macro then contains the default value which can be scanned for pointers. If the default value macro is not of the expected form, then the package just executes it without attempting to retrieve the default value or replace pointers.

### Preset system

The default value system operates when users specify keys, but no value for the keys. But the keyval package does not provide a system that assigns values to keys when keys have actually not been used at all by the user. In a lot of applications, one would like to implement default values for keys when they are not used. For instance, 'scale this figure with factor 1 unless specified otherwise by the user'. One could go ahead and call the key macro with a default value and afterwards, submit the user input to `\setkeys` and possibly overwrite the values that you have just set. This is possible (but quite cumbersome when there are many keys) in cases where keys do not generate material themselves, but, for instance, only set a length.

But what happens if we apply this scheme to keys which are defined as follows?

```
\define@key{fam}{keya}{Your input was: #1}
\define@key{fam}{keyb}{\edef\list{\list,#1}}
```

If we follow the scheme in the first example, both our default value as well as the user input (if present) will be typeset. In the second example, both the default value and the user input will be added to the list contained in `\list`.

To avoid this, xkeyval introduces the preset system. First one declares the keys that should always be assigned and their values using `\presetkeys`, for instance

```
\savekeys{fam}{head}
\presetkeys{fam}{head=red}{tail=\usevalue{head}}
```

The function of the two arguments of `\presetkeys` will become clear in a moment.

Now, when submitting user input for keys in family `fam`, the macro `\setkeys` will determine which keys will be set by the user and avoid setting them again with the preset values. Keys that are not set by the user will be set by the values specified in `\presetkeys`.

However, there is one thing that we should keep in mind in this system when pointers are used. If the pointer points to a key which is assigned a value afterwards, the pointer cannot know this value yet and errors will occur. Hence, it is best (in most situations) to execute preset pointers at the very end.

On the other hand, if a value for a key has been saved, let's say `blue`, and the user first issues a pointer to that key and later the preset value sets the key to `red`, the outcome of the pointer will of course be `blue`, which was actually not the intention when setting the preset value to `red`. Hence, for ordinary keys, it is best to execute them at the very beginning, before setting user input.

That is why the `\presetkeys` macro has two arguments: the first one (usually containing ordinary keys and values) will be inserted before setting user input keys, the second one (containing pointers to preset values or user input) afterwards.

This system is especially useful when you can't rely on key values remaining local to a macro or environment since the preset system will, at every use of your macro or environment, reset key values to the preset value unless overwritten locally by the user. This needs some more explanation. `\def` definitions (for instance made by key macros) will be destroyed by TEX when leaving a group or environment. Hence the values will remain local. However, if your keys are not using `\def`, but for instance, `\gdef`, this definition will escape the group or environment and might distort all following macros or environments. Hence, you will have to take care to reinitialize the key values at every use of the macro or environment.

This is, however, not necessary anymore with the preset system. Once the preset keys have been defined for a specific family, each time this family is used in the `\setkeys` command, the preset values will be taken into account together with the user input.

The following example will demonstrate the power

of the preset system in combination with pointers. Below the example, you can find its output and the explanation. Let's assume we want to create a simple frame/shadow box command with the following *default* behaviour:

- a shadow will be drawn if and only if the box is framed;
- the shadow color should be a 40% tint of the frame color, thus being clearly discernible;
- the shadow size (or width) should be 4 times the width of the frame.

Certainly, the user should be able to overrule each of these default parameter relations when the box command is actually applied.

```
\documentclass{article}
\usepackage{xkeyval}
\usepackage{calc,xcolor}

\makeatletter
\newdimen\shadowsize
\define@boolkey{Fbox}{frame}[true]
\define@boolkey{Fbox}{shadow}[true]
\define@key{Fbox}{framecolor}%
 {\def\Fboxframecolor{#1}}
\define@key{Fbox}{shadowcolor}%
 {\def\Fboxshadowcolor{#1}}
\define@key{Fbox}{framesize}%
 {\setlength\fboxrule{#1}}
\define@key{Fbox}{shadowsize}%
 {\setlength\shadowsize{#1}}
\savekeys{Fbox}{frame,framecolor,framesize}
\presetkeys{Fbox}%
 {frame,framecolor=red,framesize=0.5pt}%
 {shadow=\usevalue{frame},
  shadowcolor=\usevalue{framecolor}!40,
  shadowsize=\usevalue{framesize}*4}
\newcommand*\Fbox[2][]{%
 \setkeys{Fbox}{#1}%
 {\ifKV@Fbox@frame\else\fboxrule0pt\fi
  \ifKV@Fbox@shadow\else\shadowsize0pt\fi
  \sbox0{\fcolorbox{\Fboxframecolor}{white}{#2}}%
  \hskip\shadowsize
  \color{\Fboxshadowcolor}%
  \rule[-\dp0]{\wd0}{\ht0+\dp0}%
  \llap{\raisebox{\shadowsize}%
   {\box0\hskip\shadowsize}}}%
}
\makeatother

\begin{document}
\Fbox{demo1}
\Fbox[framecolor=blue]{demo2}
\Fbox[shadow=false]{demo3}
\Fbox[framesize=1pt]{demo4}
\Fbox[frame=false,shadow]{demo5}
\end{document}
```

demo1 demo2 demo3 demo4 demo5

First of all, lines 7 to 16 define the keys to be used in the example. The `\presetkeys` command in line 18 defines the presets: the frame will be set to true, its color to red and the frame size to 0.5pt, unless the user provides different specifications for these keys. The requirements listed above are then covered by the pointer expressions in the next argument.

The first box application now shows the default box without additional user input. We see a frame and a shadow, based on the color red. The second box shows that the user input for the frame color will overwrite the preset values and turn the box blue. But since the shadow color equals the frame color by default, the shadow is blue as well. In the third example, we have a frame, but no shadow. Notice that the color has returned to red, the default value. The fourth box has an increased frame size and hence an increased shadow size as well due to the pointer use when presetting the keys. The last example shows that it is possible to overwrite the default behaviour of linking shadows to frames: it displays a shadow without a frame.

## Robust parsing

Just as with the pointer delimiters `\savevalue` and `\usevalue`, keyval and xkeyval treat the comma and the equality sign as delimiters. In the past, this has led to problems. A well known incompatibility exists between the Turkish language version of the babel package and all packages using keyval. Since Turkish babel changes the catcode of the equality sign for shorthand notation, the parsing macros of keyval cannot detect these characters anymore and will generate errors. [4]

xkeyval solves this by sanitizing (i.e. setting the catcode to 12) all characters necessary to parse the input properly. This is done using the `\@selective@sanitize` macro, which can sanitize one or more different characters in a single run. Moreover, the sanitize group depth can be controlled. xkeyval implements the macro such that only commas and equality signs appearing in the top level of a key value will be sanitized, since that is all that's needed for input parsing. Characters inside groups are left untouched and can hence possibly even contain babel shorthand notation without causing errors:

```
\usepackage[turkish]{babel}
...
\setkeys{fam}{key={some =text}}
```

In this example, the first '=' will be sanitized for parsing, whereas the second '=' will be left untouched and thus keeps its original meaning.

### Redefining macros?

Obviously, redefining existing macros is dangerous in general. Still the xkeyval package redefines the two major keyval macros `\define@key` and `\setkeys`. The reason is that this will avoid any confusion of having several systems running next to each other, doing approximately the same things.

Although xkeyval supports all of the originally possible syntax of the keyval package, we still had to check the packages using keyval before we could make the decision to redefine the macros. Three major issues came up in that process.

First of all, we found that some packages were using keyval internals directly instead of the user interface formed by `\define@key` and `\setkeys`. To avoid any errors of undefined control sequences in these packages, xkeyval loads the keyval internals if keyval hasn't been loaded before.

Secondly, certain packages implemented a creative use of the default value system as has been discussed in the section about the pointer syntax. The solution of xkeyval has also been discussed there.

Finally, we found that the pst-key package was redefining `\define@key` and `\setkeys` itself to provide the means of setting PSTricks keys. After discussing this with the PSTricks maintainer Herbert Voß, we agreed that xkeyval would develop a unified approach to keys and values and that the pst-key package would be abandoned. More information on the development related to PSTricks is provided in the final section of this article.

After redefining the necessary macros, xkeyval will make sure that the keyval package cannot be loaded anymore in order to avoid again redefining the xkeyval macros. This was the final step necessary in safely redefining the keyval macros and to provide a system which all package authors can convert their package to without too much effort.

### The pst-xkey package

An important stream of packages will be using xkeyval already in the near future. These are the PSTricks packages [3, 4], currently relying for key and value processing on a combination of private definitions in `pstricks.tex` and pst-key, the latter being a modification of the keyval package.

Due to the popularity and flexibility of the PSTricks package, several people have contributed extensions to the original distribution. Unfortunately, all PSTricks keys have the standard form `\psset@somekey`, thus package authors need to check all existing packages to be sure not to redefine one of the existing keys.

The PSTricks maintainer Herbert Voß has recognized this problem and soon the work on xkeyval started to provide a way to define and set PSTricks keys via this package. The major advantage would be the possibility for individual package authors to nest their keys in a well chosen family (for instance, the package name) and avoid the need to check other packages for existing keys.

In order to make this possible, `\define@key` and `\setkeys` needed to be adjusted a bit. Further, the `\psset` macro needed to be redefined to use the new `\setkeys` and let this scan all families available. When a PSTricks package is loaded, it adds all families used in the package to a list and this list will be used in `\setkeys`. Since all separate packages will use different families, reusing key names is not a problem anymore. The redefinition of `\psset` and some other macros necessary to do the job, is available in the `pst-xkey` package which comes with the xkeyval package.

Due to the vastness of the PSTricks collection of packages, the conversion of all packages to use pst-xkey instead of pst-key will take some time, but will be done in the near future.

### Footnotes

1. Note that `author=\protect\textit{Me}` is *no* solution for this problem.
2. Please refer to the documentation of the xkeyval package to learn about further syntactical details which are not discussed in this article.
3. Except if the pointer is hidden for xkeyval inside a group.
4. See for more information concerning this problem of keyval and babel: `http://www.latex-project.org/cgi-bin/ltxbugs2html?pr=babel/3523`

### References

[1] Hendri Adriaens. xkeyval package, v1.8c, 2005/01/01. `CTAN:/macros/latex/contrib/xkeyval`.

[2] David Carlisle. keyval package, v1.13, 1999/03/16. `CTAN:/macros/latex/required/graphics`.

[3] Herbert Voß. PSTricks website. `http://www.pstricks.de`.

[4] Timothy Van Zandt et al. PSTricks package, v1.04, 2004/06/22. `CTAN:/graphics/pstricks`.

Hendri Adriaens
`http://stuwww.uvt.nl/~hendri`
Uwe Kern
`http://www.ukern.de`