Macros

Implementing key–value input: An introduction

Joseph Wright and Christian Feuersänger

Abstract

The key–value system is justly popular as it greatly simplifies controlling packages for the user. Unfortunately, that ease of use is not transferred into setting up key–value systems for authors of pre-packaged TeX code. This article describes how to implement key–value controls for both TeX and LaTeX authors, including a brief overview of how the underlying system works. As well as the original keyval package, the various extended keyval-based packages are covered, looking at the relative advantages of each system. Looking beyond keyval-based systems, an overview of the pgfkeys package is also given.

1 Introduction

The key–value method uses a comma-separated list of ⟨*key*⟩=⟨*value*⟩ to set one or more ⟨*key*⟩s. The code applied when a ⟨*key*⟩ is given can undertake a range of processing on the ⟨*value*⟩. Almost every (LA)TeX user will have come across the power of the this method for providing control values. The interface is increasingly widespread in controlling package and class behaviour. It offers a much cleaner method for managing large numbers of options or control values than defining multiple single-use macros and complex optional arguments.

The original keyval package (Carlisle, 1999) provides a core of functionality. This has been extended by xkeyval (Adriaens, 2008), kvoptions (Oberdiek, 2009a) and kvsetkeys (Oberdiek, 2009b), providing additional tools for the developer, and making key–value input available for LaTeX package and class options.

Unfortunately, the ease of key–value input for the user has not translated into easy development of new uses of key–value syntax in package control. Many (even experienced) (LA)TeX code authors struggle to make a start with implementing key–value methods. This article aims to make key–value input more accessible. The major use of key–value syntax is controlling LaTeX packages and classes, and this is reflected in the focus here. However, all of the key–value implementations are compatible to some extent with plain TeX. A short section on use with plain TeX is included here, and as far as possible all of the examples use only plain TeX macros. Throughout the article, "package" is used to refer to a LaTeX package, LaTeX class or other file using key–value input.

The pgfkeys system implements a key–value interface in a somewhat different manner from the various keyval-derived packages. As a result, it has unique strengths. Due to the differing approaches of the keyval-based systems and pgfkeys, the latter is covered in its own section. Many of the concepts from the keyval package and its derivatives apply to pgfkeys, and so the general introduction is useful even for users who have already decided on pgfkeys.

The various packages discussed have a range of features not covered in this article: in order to remain accessible, only the most widely-applicable concepts are discussed. Some simplifications have also been made where these will not impede the more advanced user. More detail can of course be found in the various package documentation. There is also a *TUGboat* article covering the design and some of the more advanced features of xkeyval (Adriaens and Kern, 2005).

2 How key–value works

There are two parts to using the key–value system: defining keys, and assigning values to keys. When using the keyval package itself, these tasks are handled by the macros `\define@key` and `\setkeys`, respectively.

The *key* in key–value input is the "name" of a data item. The model used by keyval divides keys into *families*: groups of keys that can be processed together. The `\define@key` macro is used to define keys. This requires three pieces of information: the key name, the family to which the key belongs, and a handler for the key. Consider a package `fam` defining a key `key`, which simply prints the value given:

```
\define@key{fam}{key}{#1}
```

As can be seen, `\define@key` takes three arguments, ⟨*family*⟩, ⟨*key*⟩ and ⟨*handler*⟩. The ⟨*handler*⟩ receives the value given for the key as macro argument `#1`, and can consist of any TeX code appropriate to process the *value* assigned to the key (the part after the equals sign).

How does `\define@key` work? A new macro \⟨*prefix*⟩`@`⟨*family*⟩`@`⟨*key*⟩ is defined, with expansion ⟨*handler*⟩. So in the example above, the following would achieve the same effect:

```
\def\KV@fam@key#1{#1}
```

Here, the *prefix* is a code added to the beginning of the key name, and acts as a family of families. The prefix is fixed with the value `KV`: only xkeyval allows this to be varied.

The `\setkeys` macro is then used to set key values, the second part of the key–value concept. The input to `\setkeys` is a comma-separated list: each comma-separated ⟨*key*⟩=⟨*value*⟩ pair is therefore processed in turn. Unlike the majority of TEX macros, this process ignores spaces between key–value pairs:

```
\setkeys{fam}{
  key one=value 1 ,
  key two=value2
}
```

consists of two key–value pairs "`key␣one=value␣1`" and "`key␣two=value2`". Notice that both the key name and the value can contain spaces. Braces must be used to protect literal "`,`" and "`=`" characters inside `\setkeys`:

```
\setkeys{fam}{
  key three={value1,value2},
  key four={some=stuff}
}
```

For each pair found, `\setkeys` then attempts to separate the data into a key and a value, delimited by an equals sign. If there is no equals sign, an error will normally be raised. Assuming a value is found (even an empty one, if there is nothing after "`=`"), `\setkeys` looks for a macro of the form \⟨*prefix*⟩`@`⟨*family*⟩`@`⟨*key*⟩ to handle the input. If such a macro exists, it is executed with the value as argument `#1`. If no macro is found, the key is regarded as undefined, and an error is raised. In the example earlier, the result of the `\setkeys` operation is to supply the key macro for `key one` with "`value 1`", and that for `key two` with "`value2`".

`\setkeys` passes the value to the processing macro as is. Thus macro names, *etc.*, can be used without worrying about expansion in the process.

3 Defining keys

As outlined in the previous section, a key is defined by creating a suitably-named macro. However, defining every key using `\def` or `\newcommand` would add considerably to the effort of using key–value input. All of the packages discussed here provide more convenient methods.

3.1 Using the keyval package

The `\define@key` macro for key definition is the only method that the original keyval package provides. However, this is the most powerful method for defining a key: the developer is completely free to code any handler required. One particularly common process is to store the value in a macro to be used later:

```
\define@key{fam}{key}{\def\fam@data{#1}}
```

This stores the value given for `key` in `\fam@data`. The definition of the storage macro does not occur until the key is used for the first time. Thus if the macro must be defined even if the key has not been used, an additional line is necessary:

```
\def\fam@data{initial}
\define@key{fam}{key}{%
  \def\fam@data{#1}%
}
```

Setting the `key` key will then redefine `\fam@data` to contain whatever value is passed to the key. Notice that here the key family has been used as the start of the storage macro name.

As was explained in Section 2, keys must have a value (even if this is empty). It is possible to specify a default value for a key, which is then used if the user does not supply one (this does *not* mean that the key is defined before it is first used!). A default value is supplied as an optional argument to the `\define@key` macro, which following the LATEX convention appears in square brackets:

```
\define@key{fam}{key}[default]{%
  \def\fam@data{#1}%
}
```

This means that

```
\setkeys{fam}{key}
```

is interpreted as though the user had written

```
\setkeys{fam}{key=default}
```

The handler macro receives the default value in exactly the same way as user-supplied data.

Using the "raw" `\define@key` macro rapidly becomes awkward when a large number of similar keys are required. Package authors can of course write short-cut macros to make the process easier. However, the other key–value packages seek to address this issue by making one or more common key definitions available directly.

3.2 Using kvsetkeys

Using kvsetkeys adds several "low-level" functions to keyval; those related to setting keys will be addressed later. kvsetkeys does not add any methods for processing *known* key names, and indeed relies on the explicit loading of keyval to define keys. It does, however, add a customised handler for key names which have not been defined.

When using the kvsetkeys package, a handler for unknown keys in a family is created using the macro `\kv@set@family@handler`. This allows data input for arbitrary key names, or perhaps simply a customised warning or error message. The name of the key used is available as `#1`. A simple warning could be given by:

```
\kv@set@family@handler{fam}{%
  \wlog{Warning: key '#1'
    unknown by package fam}
}
```

A more complex example might be to use the input to define a new macro. The value given for the key (if any) is available as `#2`. For example,

```
\kv@set@family@handler{fam}{%
  \expandafter\def\csname
    fam@user@#1\endcsname{#2}%
}
```

creates a new internal macro including the name of the unknown key to store the given value. Notice that the definition includes a marker that this is a user-provided key name (`\fam@user@`), as no check has been made for an existing definition.

3.3 Using kvoptions

As the package name indicates, kvoptions helps LaTeX developers use key–value input for package and class options. However, as we will see later, there is no fundamental difference between defining keys and defining key–value package options.

The kvoptions package makes life easier for the author by allowing the family value to be defined once, and then used in all subsequent key definitions. It also automatically generates various macros for the package author:

```
\SetupKeyvalOptions{
  family = fam,
  prefix = fam@
}
```

This defines the family as `fam`, and prefixes all new storage macros with `\fam@`. This does *not* affect the key prefix, used for the key macros themselves, which still start with `\KV@...`. Usually, the ⟨*prefix*⟩ given here will be simply ⟨*fam*⟩`@`, as this means all storage macros are defined as `\fam@...`. The rest of this section assumes this convention is used, and that the setup above applies. If no data has been supplied using `\SetupKeyvalOptions`, the family and macro prefix are taken from the name of the current package.

The kvoptions package provides macros for defining new keys (or options):

- `\DeclareBoolOption`;
- `\DeclareComplementaryOption`;
- `\DeclareStringOption`.

The names of the macros are a good guide to the general method key type they produce. kvoptions also provides methods applicable only to package options: these are discussed later.

`\DeclareBoolOption` creates a true/false key. Giving the key name alone is the same as giving it with the `true` value. A new switch is created which is named `\if`⟨*fam*⟩`@`⟨*key*⟩, which works in the same way as though created using `\newif`.

```
\DeclareBoolOption{active}
% Other code
\iffam@active
  % Do stuff
\else
  % Do nothing
\fi
```

`\DeclareComplementaryOption` creates a complementary key to an existing Boolean key. The most common example might be setting draft *versus* final:

```
\DeclareBoolOption{final}
\DeclareComplementaryOption
  {draft}{final}
% Other code
\iffam@final
  % Do final stuff
\else
  % Do draft stuff
\fi
```

In this way, the same switch may be set by keys with differing names.

`\DeclareStringOption` creates a new storage macro, to hold the data provided as the key value. This is similar to the `\define@key` method for saving to a macro given earlier.

```
\DeclareStringOption{key}
```

stores the value given in the macro `\fam@key`. An initial value can be provided for the option, so that `\fam@key` will be defined under all circumstances. This uses a LaTeX optional argument;

```
\DeclareStringOption[initial]{key}
```

has a similar result to

```
\def\fam@data{initial}
\define@key{fam}{key}{%
  \def\fam@key{#1}%
}
```

so that `\fam@key` will expand to "initial", until the key is set to an explicit value.

3.4 Extended keyval: xkeyval

The xkeyval package extends the key–value system further than any of the other packages. As a result, it has a much richer (and more complex) command syntax. The first point to note is that, unlike the other packages discussed, xkeyval allows the developer to alter the key prefix. This is achieved by adding an optional argument to `\define@key`:

```
\define@key{fam}{key}{#1}
\define@key[pre]{fam}{key}{#1}
```

The first command defines `\KV@fam@key` as the key-handling macro; the second defines `\pre@fam@key`.

If no explicit key prefix is given, the value KV is used. Of course, altering the key prefix means that \setkeys also needs to be modified to accommodate it. To set the two keys above, the appropriate \setkeys commands would be

```
\setkeys{fam}{key=input}
\setkeys[pre]{fam}{key=input}
```

Notice that, in contrast to kvoptions, there is no method to pre-set the family, *etc.* As a result, when defining a large number of keys it is often convenient to first create customised definition macros:

```
\def\fam@define@key{\define@key{fam}}
\def\fam@define@mykey
  {\define@key[pre]{fam}}
```

As is the case with kvoptions, xkeyval provides an extended set of key definition types:

- \define@key;
- \define@boolkey;
- \define@boolkeys;
- \define@cmdkey;
- \define@cmdkeys;
- \define@choicekey.

The extended version of \define@key has already been discussed. The concept of key prefix applies to all of the other key types, although the remaining examples all use the default KV prefix. If the prefix is given, it is always the first, optional, argument to the definition macro.

The \define@boolkey macro creates a single Boolean key. The key definition requires a function, even though this may be blank. To allow the key name alone to be used as equivalent to key=true, a default value is needed. This follows the LaTeX convention of appearing in square brackets, but is not the first argument given: instead, it follows the key name, for example,

```
\define@boolkey{opt}{key}[true]{}
```

creates a new switch \ifKV@fam@key, and a key-processing macro \KV@fam@key with no customised function attached: the \if is simply set appropriately. The name of the new switch can be altered using a second option argument to specify the macro prefix. This again appears in square brackets, between the family and key names:

```
\define@boolkey{opt}[fam@]
  {key}[true]{}
```

creates the switch \iffam@key, and is functionally equivalent to the \DeclareBoolOption macro from kvoptions.

Several Boolean keys can be created in one go using \define@boolkeys. Here, no custom function is needed (or indeed permitted). A default value is still needed to allow use of the key name alone:

```
\define@boolkeys{opt}[fam@]
  {key,key two,key three}[true]
```

Using \define@cmdkey creates a storage macro for the value given, along with a processing macro. This can become somewhat complicated, and so some examples are needed.

```
\define@cmdkey{fam}{key}{}
```

creates a new key macro \KV@fam@key, which will store the input in \cmdKV@fam@key. The name of the storage macro can be altered by adding a macro prefix argument, as with Boolean keys:

```
\define@cmdkey{fam}[fam@]{key}{}
```

The name of the *key* macro is unchanged, but the storage macro is now called \fam@key. Notice that both examples include a final processing argument: in these examples this is blank as storage of the input alone is required. A default can be given for a command key, as an optional argument after the key name:

```
\define@cmdkey{fam}[fam@]{key}
  [default]{}
```

The \define@cmdkeys macro allows the creation of several keys at one go, using a comma-separated list. Only one default is available for all of the commands, and a custom function cannot be given. In many cases, this will not be an issue as the stored value is the aim of the key. For example, to create three command keys key, key two and key three:

```
\define@cmdkeys{fam}[fam@]
  {key,key two,key three}
```

For large numbers of storage keys, this method is preferable to multiple calls to \define@cmdkey.

Finally, \define@choicekey allows creation of a key with a limited number of valid input values from an arbitrary list. This key type has several optional arguments which make it somewhat difficult to set up without experimentation. At the most basic, the value is checked by xkeyval and is then passed to key handler function:

```
\define@choicekey{fam}{key}
  {val1,val2,val3}
  {You chose: #1}
```

Here, the key key can take only the values val1, val2 and val3. The * modifier makes the comparison by \define@choicekey case-insensitive.

```
\define@choicekey*{fam}{key}
  {val1,Val2,VAL3}
  {You chose: #1}
```

will match key=val1, key=Val1, *etc.* In these examples, the processing macro simply displays the user's choice. Further processing of keywords is possible in this argument, for example to set several switches based on a keyword. Adding the + modifier to \define@choicekey makes a second handler available for items not on the list:

```
\define@choicekey+*{fam}{key}
  {val1,val2,val3}
  {You chose: #1}
  {\wlog{Invalid choice '#1': you
    must put 'key=val1', 'key=val2'
    or 'key=val3'}%
  }
```

Here, valid choices act as in the previous example. Any other value will use the second handler, which in this case simply writes a warning to the log.

The macros outlined above all have more extended syntax, with additional optional arguments. This more complex area has been covered by the authors of xkeyval (Adriaens and Kern, 2005).

4 Setting keys: user interface

As described in Section 2, the keyval package sets key values using the \setkeys macro. The same is true for kvoptions and xkeyval (the latter overloads its own modified version of the macro). In contrast, kvsetkeys uses the \kvsetkeys macro; this is designed to be more robust than \setkeys as defined by keyval, and to cope better with altered catcodes for "," and "=".

The \kvsetkeys macro can also set keys from the other packages, provided they use the key prefix KV. Thus the only keys that cannot be set by \kvsetkeys are those produced using xkeyval with a non-standard key prefix. In the following discussion, \setkeys could therefore be replaced by \kvsetkeys.

The \setkeys macro needs to know the family (and potentially prefix) to which keys belong. Often, and especially when developing a package, a user macro which already contains this information is desirable. The usual method is to define a custom setup macro:

```
\def\famsetup#1{\setkeys{fam}{#1}}
```

An optional key–value argument to user macros is often defined, so that settings apply only to that instance of the macro. Provided the processing of the macro occurs inside a group, this is easy to achieve (using LaTeX for convenience):

```
\newcommand*{\mycmd}[2][]{%
  % #1 is the optional keyval argument
  % #2 is a mandatory argument
  \begingroup
    \setkeys{fam}{#1}%
    % Do stuff with #2
  \endgroup}
```

4.1 \kvsetkeys *versus* \setkeys

Using \kvsetkeys adds three major refinements to the keyval \setkeys macro. Firstly, \kvsetkeys reliably sets keys when the catcodes for "," and "=" are non-standard. This is important when using packages that make the equals sign active, for example the turkish option of babel. The xkeyval version of \setkeys also handles these cases correctly.

Secondly, both \kvsetkeys and \setkeys remove some braces from value input. \kvsetkeys aims to be more predictable. It removes only one set of curly braces, whereas \setkeys may remove one or two sets of braces, depending on circumstances.

Finally, \kvsetkeys supports the unknown key handler. This will be many authors' motivation to use kvsetkeys: handling unknown keys otherwise requires adding custom low-level code.

5 LaTeX package and class options

The preceding sections apply to using key–value methods in a wide variety of situations. One of the most common aims of authors considering key–value input is to use it for processing LaTeX package or class options. This has particular points to consider, and therefore specialised macros have been made available for this area.

Any key defined when processing occurs is available as an option. This means that options can be created using \define@key or any of the higher-level macros listed here. It also means that any key–value option is also a valid key. This may not always be desirable, and is considered further in Section 6.1.

Before using key–value options, the careful developer should know the limitations of the system. Before package options are passed to the key–value system, they are processed by LaTeX. The kernel removes all unprotected spaces from the input, which means that key names' spaces will be rendered useless. Secondly, unlike direct use of \setkeys, the kernel will expand the input. This means that some keys should *not* be given as options to a package.

Although patches exist to deal with these problems, these are not generally useful: the patches must be loaded before input of the package or class requiring them! This leaves the package author with two options. The first approach is to abandon key–value load-time options, with a setup macro used only after loading the package. More commonly, the options can be designed to minimise the impact of the problem. Design steps to achieve this include:

- Avoiding any key names containing spaces;
- For keys which will receive values containing spaces, initially defining the key to gobble the

value with a warning, then redefining it after processing options to the real meaning (see Section 6.1);
- For keys that will require a single macro, requiring the csname rather than the macro itself, then using `\csname...\endcsname` in the implementation.

To allow key–value syntax to be used in package options, the standard LaTeX method for handling option input has to be modified. This can be done directly, but copy–pasting code is not normally considered good programming. xkeyval and kvoptions both provide suitable macro definitions.

5.1 Using kvoptions

When using kvoptions, option processing takes place using the `\ProcessKeyvalOptions` macro. This has to be supplied with the family of keys to be processed:

```
\ProcessKeyvalOptions{fam}
```

To make handling certain styles of option easier, kvoptions provides two key-defining macros which are very focussed on package options. Options acting in the normal LaTeX manner are created by the `\DeclareVoidOption` macro. The key is to be used alone, but if a value is given it is ignored with a warning. As this is essentially a standard LaTeX option, the normal need to provide an action exists:

```
\DeclareVoidOption{old}{%
  \PackageInfo{fam}{You gave the ‘old’ option}%
}
```

`\DeclareDefaultOption` is used to process unknown options, in the manner of the LaTeX kernel `\DeclareOption*` macro. The result is that `\CurrentOptionKey` stores the current key name, with `\CurrentOptionValue` holding any value which was given, or `\relax` if there is no value.

```
\DeclareDefaultOption{%
  \PackageInfo{fam}{%
    You gave the ‘\CurrentOptionKey’ option,
    with value ‘\CurrentOptionValue’
  }%
}
```

5.2 Using xkeyval

The `\ProcessOptionsX` macro is used to process xkeyval options. As might be expected, this takes an optional prefix and mandatory family argument. The family has to be given in angle brackets, for example

```
\ProcessOptionsX<fam>
```

Loading xkeyval provides `\DeclareOptionX` for handling package options which may have no value. Values *can* be accepted, and are available as `#1`. This macro does not require a key family, although one can be given as an optional argument, again in angle brackets.

```
\DeclareOptionX<fam>{letter}{%
  \PassOptionsToPackage{geometry}
    {letter}%
}
\DeclareOptionX<fam>{date}
  {\renewcommand*{\date}{#1}}
```

The `\DeclareOptionX*` macro works like the kernel's `\DeclareOption*` macro, but no error is raised if the option is in ⟨*key*⟩=⟨*value*⟩ format. In contrast to kvoptions, the entire unknown input (key, plus potentially an equals sign and a value) is stored as `\CurrentOption`.

```
\DeclareOptionX*{%
  \PackageWarning{fam}
    {‘\CurrentOption’ invalid}}
```

6 Additional considerations

6.1 Redefining and disabling keys

Keys can be (re)defined at any point using any of the key-defining macros discussed here. Thus keys can be defined to only give a warning, then redefined later to carry out a function. This is particularly useful for LaTeX package options, where the key may not be appropriate at load time but may be later.

Conversely, some keys are appropriate only before some action (such as loading a file) takes place. Disabling a key simply requires that the key is defined to do nothing:

```
\define@key{fam}{key}{\wlog{Key ‘key’ ignored}}
```

If a key (re)definition occurs inside a group (such as `\begingroup...\endgroup` or `{...}`), the definition applies only inside that group. There is no `\global` prefix to `\define@key`, and so to ensure that a key is globally disabled, the low-level TeX `\gdef` must be used:

```
\gdef\KV@fam@key#1{\wlog{Key ‘key’ ignored}}
```

Both kvoptions and xkeyval provide high level methods for disabling keys. kvoptions defines the `\DisableKeyvalOption` macro, which requires only the family and key name:

```
\DisableKeyvalOption{fam}{key}
```

This macro takes an optional argument which can be used to control the result of attempting to use a disabled key (warning, error, ignore, *etc.*). The use of the optional argument is illustrated in Section 7. xkeyval provides the similar `\disable@keys`:

```
\disable@keys{fam}{key}
```

In this case, the macro can accept the usual xkeyval optional argument for the key prefix.

6.2 Setting one key from another

There are occasions when the setting of one key affects another. Usually, this can be accommodated using `\setkeys` within `\define@key` (or a derivative, if using xkeyval):

```
\define@key{fam}{key}{#1}
\define@key{fam}{key two}{%
  You said: \setkeys{fam}{key=#1}%
}
```

If two keys should function in an identical manner, it is sometimes easier to `\let` one to the definition of the other. Be careful about default values: only the key defined using `\define@key` will have one using this method! This issue can be avoided by first declaring the keys as normal, then carrying out the `\let`.

```
\define@key{fam}{key}[default]{#1}
\define@key{fam}{key two}[default]{}
\expandafter\let\csname
  KV@fam@key two\endcsname\KV@fam@key
```

gives two identical keys, `key` and `key two`, with the same default.

The use of these methods to allow alternative spellings for setting a key, to set a storage macro and a TeX `\if...`, are illustrated in Section 7.

6.3 Interaction between the different key–value packages

The xkeyval, kvoptions and kvsetkeys packages all use unique macro names (both user and internal). All three can therefore be loaded without issue. Provided the standard key prefix `KV` is used, the keys generated are also cross-compatible.

Neither kvoptions nor kvsetkeys define any of the macros from the keyval package itself. This means that they require keyval, and that they do not affect its functions. xkeyval works differently, using its own definition of the core keyval macros, and under LaTeX prevents subsequent loading of the keyval package. xkeyval aims to make these changes backward-compatible; however, under certain circumstances some macros may behave differently. The latest version of xkeyval fixes a number of differences in behaviour between keyval and xkeyval.

The following short LaTeX document can be used as a test to show the differences in behaviour between older versions of xkeyval and the keyval package. With keyval or the latest version of xkeyval this document compiles correctly. However, older versions of xkeyval give errors.

```
\documentclass{article}
\usepackage{keyval}
%\usepackage{xkeyval}
\makeatletter
\define@key{w}{cmd}
  {\def\test##1{#1}}
\makeatother
\setkeys{w}{cmd={--#1--}}
\begin{document}
[\test{ee}]
\end{document}
```

It is therefore strongly recommended that any package using key–value should be tested with xkeyval loaded, even if it is not being used. In this way, if other packages load xkeyval problems should be avoided.

6.4 Using key–value with plain TeX

All of the key–value packages are compatible to some extent with plain TeX. Both kvoptions and kvsetkeys are designed to auto-detect whether TeX or LaTeX is in use. A minimal set of LaTeX macros are defined only if they are not otherwise available. Thus both can be used directly in plain TeX.

```
\input kvoptions.sty
\input kvsetkeys.sty
```

The xkeyval bundle is designed in a modular fashion. The file `xkeyval.sty` contains the LaTeX code (including processing code for package options), whereas the code for defining and setting keys is contained in `xkeyval.tex`. As plain TeX users need only the latter, using xkeyval is simply:

```
\input xkeyval
```

The keyval package itself is not designed for use with plain TeX. It therefore requires a small but non-zero number of LaTeX macros. These are conveniently provided by miniltx.

```
\input miniltx
\input keyval.sty
```

The file `keyval.sty` is also loaded by kvoptions, which ensures that the necessary macros are defined.

7 Putting it all together: a short example

The various methods outlined above will be sufficient for many people implementing a key–value interface. However, putting everything together can still be challenging. A short, and not entirely trivial, example will illustrate the steps needed.

Consider the following situation. You have been asked by an inexperienced LaTeX user to produce a small package providing 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 of all of these. This should be controllable on loading the package, or during the document. Finally, a de-activation setting is requested, so that `\xmph` 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 at the problem, you first decide to call the package xmph, and to use the xmph@ prefix for internal macros. The settings requested all look relatively easy to handle using the kvoptions package, so you choose that for key–value support. You decide on the following options/settings:

- inactive, a key with no value, which can be given only in the preamble;
- useitalic, a Boolean option for making the text italic;
- usebold and usecolour, two more Boolean options with obvious meanings
- colour, a string option to set the colour to use when the usecolour option is true.

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

As well as the \xmph macro, you decide to create a document body setup macro \xmphsetup. Both \xmph and \xmphsetup will take a single, mandatory argument. This keeps everything easy to explain, and means there is not too much work to do with arguments and so on.

With the design decisions made, you can write the package. The options and so on come first. Most of the keys are defined using high-level kvoptions macros, although two low-level methods are used. Initial settings for the package are set up by a \setkeys instruction *before* processing any package options.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{xmph}
  [2008/03/17 v1.0 Extended emph]
\RequirePackage{color,kvoptions}
\SetupKeyvalOptions{
  family=xmph,
  prefix=xmph@}
\DeclareBoolOption{useitalic}
\DeclareBoolOption{usebold}
\DeclareBoolOption{usecolour}
\DeclareBoolOption{usecolor}
\let\KV@xmph@usecolor
  \KV@xmph@usecolour
\DeclareStringOption{colour}
\define@key{xmph}{color}
  {\setkeys{xmph}{colour=#1}}
\DeclareVoidOption{inactive}{%
  \PackageInfo{xmph}
    {Package inactive}%
  \AtEndOfPackage{\let\xmph\emph}%
}
\setkeys{xmph}{useitalic,colour=red}
\ProcessKeyvalOptions{xmph}
\define@key{xmph}{inactive}
  {\PackageInfo{xmph}
     {Package inactive}
  \let\xmph\emph
}
\AtBeginDocument{
  \DisableKeyvalOption[
    action=warning,
    package=xmph]
    {xmph}{inactive}
}
\newcommand*{\xmphsetup}
  {\setkeys{xmph}%
}
```

The user macros are then defined; by keeping the two parts separate, it will be easier to alter the method for managing the keys, if needed. Later, we will see how this enables switching from keyval-based keys to pgfkeys without altering the core of the package at all.

```
\newcommand*{\xmph}[1]{%
  \xmph@emph{\xmph@bold{%
    {\xmph@colourtext{#1}}}}%
}
\newcommand*{\xmph@emph}{%
  \ifxmph@useitalic \expandafter\emph
  \else \expandafter\@firstofone
  \fi}
\newcommand*{\xmph@bold}{%
  \ifxmph@usebold \expandafter\textbf
  \else \expandafter\@firstofone
  \fi}
\newcommand*{\xmph@colourtext}{%
  \ifxmph@usecolour \expandafter\textcolor
  \else \expandafter\@secondoftwo
  \fi
  {\xmph@colour}}
```

The actions of the new package are shown by the following short example LaTeX file. The use of the disabled key inactive will result in a warning entry in the log.

```
\documentclass{article}
\usepackage[
  usecolour,
  usebold]{xmph}
\begin{document}
  Some text \xmph{text}
  \xmphsetup{
    usecolor=false,
    usebold=false,
```

```
    useitalic=false}%
  \xmph{more text}
  \xmphsetup{inactive}
\end{document}
```

8 A different approach: pgfkeys

All of the packages discussed so far are built on the keyval approach. Keys are part of families, and further subdivision (at least beyond altering the key prefix) is not readily achieved. An alternative approach is taken by the pgfkeys package (Tantau, 2008). This package uses the ⟨*key*⟩=⟨*value*⟩ input format, but the underlying implementation is not derived from keyval; the pgfkeys package therefore uses a unique key management model. Thus, while for the user pgfkeys and keyval are very similar, for the developer they require different approaches. However, many of the ideas of keys with differing behaviours carry through from the earlier discussion.

8.1 How key–value works with pgfkeys

In principle, pgfkeys works in the same ways as described in Section 2: there are two parts of the key–value system, defining keys and assigning values to keys. However, pgfkeys requires just one command for both parts: the `\pgfkeys` macro.

The definition requires the use of special suffixes, the so-called key handlers. Here, the term *handler* is used slightly differently than in the other packages. For example, the statement

```
\pgfkeys{/path/key/.code={#1}}
```

defines a key named `/path/key`. The `.code` statement defines a macro which expands to the TeX code in the arguments (in our case, the TeX code is simply the argument itself, "`#1`"). Hence, using the key will just print its value:

```
\pgfkeys{/path/key=value}
```

yields "`value`". The `/path` plays a similar role to ⟨*prefix*⟩ and ⟨*family*⟩ for keyval and friends: it associates `key` with a sub-tree.

As with the key–value syntax in Section 2, spaces in key and path names are allowed, and spaces between keys and their values and different keys are ignored. Also, literal "`,`" and "`=`" characters need to be protected by braces:

```
\pgfkeys{
  /path/key three={value1,value2},
  /path/keyfour={some=stuff}
}
```

In contrast to keyval and friends, pgfkeys uses a different concept to manage key prefixes and key suffixes: the key *tree*.

8.2 The key tree

In the pgfkeys model, keys are organised hierarchically, similar to the Unix file system; subdivisions are generated using slashes. For example, `/path/sub/key` is a key named `key`, which belongs to the subtree `/path/sub` which is in turn located inside `/path`. The slash "`/`" defines the tree's root. A statement like

```
\pgfkeys{
  /path/sub/key = value,
  /path/key two = value2
}
```

sets both of these keys, showing that keys belonging to different subtrees can be set in one statement.

It is not necessary to fully qualify keys: a default path is considered for every key without a full path. For example,

```
\pgfkeys{
  key = value of key,
  key two = value of key two,
  sub/key three = value3
}
```

will search for `key`, `key two` and `sub/key three` in the current default path. Default paths can be set using a *change directory* command, using the `.cd` handler which will be discussed below. The initial setting is "`/`", which means any unqualified key name like `key` will be changed to `/key` implicitly.

8.3 Using pgfkeys

In contrast to the keyval approach, pgfkeys uses a single macro to define and set keys, namely `\pgfkeys`. At its heart, pgfkeys works with three different types of keys: keys which store their values directly, command keys and keys which are handled. Key definitions, assignments and other key types are composed of these three building blocks.

Key type 1: direct keys

Direct keys simply store their values as character sequences. A pgfkeys direct key is thus similar to a xkeyval command key (one defined using `\define@cmdkey`). For example,

```
\pgfkeys{/path/key/.initial = value}
```

defines the key `/path/key` and assigns `value`. After this, the value can be changed with assignments:

```
\pgfkeys{/path/key = new value}
```

Direct keys are stored in a way which is not directly accessible to end users. Instead, the command `\pgfkeysgetvalue` is used to get a direct key's current value into a (temporary) macro. For example, the statement

```
\pgfkeysgetvalue{/path/key}{\macro}
```

will get the current value of /path/key and copy it into \macro. The macro will be (re-)defined if necessary without affecting the stored key's value.

Putting these things together, direct keys can be used as in the following example. The code

```
\pgfkeys{/path/key/.initial = value}
\pgfkeysgetvalue{/path/key}{\macro}
After definition: ‘‘\macro’’.

\pgfkeys{/path/key = new value}
\pgfkeysgetvalue{/path/key}{\macro}
After setting: ‘‘\macro’’
```

will define /path/key with an initial value, copy the value to \macro and typeset the result. Afterwards, it changes the current value, copies the new value to \macro and typesets it again. Here's the output:

```
After definition: ‘‘value’’.
After setting: ‘‘new value’’.
```

Key type 2: command keys

The second type of pgfkeys-keys are command keys. Here, pgfkeys uses a slightly different terminology than keyval. Command keys with pgfkeys are very similar to the keys defined by \define@key: they are TEX commands with (usually) one argument replacing "#1" with the assigned value. So, what pgfkeys calls a "command key" is a "key handler" in the terminology of keyval and friends.

The usual way to define command keys is to append /.code={⟨*TEX code*⟩} to the key's name. Thus,

```
\pgfkeys{/path/cmd key/.code = {(value=#1)}}
```

defines a command key /path/cmd key which typesets "(value={⟨*its value*⟩})" whenever it is assigned. For example, the listing

```
\pgfkeys{/path/cmd key/.code = {(value=#1)}}
\pgfkeys{/path/cmd key=cmd value}
```

yields "(value=cmd value)".

As with direct keys, command keys are stored in a manner which is not directly accessible by end users. In fact, pgfkeys creates a temporary macro with \def and stores this macro into a direct key /path/cmd key/.@cmd whenever it creates a new command key.

So, command keys are TEX macros which operate on some input argument (the value) using "#1". Useful examples of command keys are

```
\pgfkeys{/path/store key/.code =
    {\def\myPkgOption{#1}
}
```

to store the input into a macro \myPkgOption or

```
\pgfkeys{/path/call key/.code = {\call{#1}}}
```

to invoke another macro \call{#1} with the value. These keys can be used with

```
\pgfkeys{
  /path/store key = value,
  /path/call key = value2
}
```

Since some processing methods are generally useful, pgfkeys provides easier ways to assign them. For example, our example of a command key which simply stores its value into a macro can equivalently be defined using

```
\pgfkeys{
  /path/store key/.store in=
    \myPkgOption
}
```

The suffix .store in, and also the suffix .code, are *key handlers*, the third type of pgfkeys options.

Key type 3: handled keys

The third type of pgfkeys-keys are handled keys.[1] If \pgfkeys encounters a key which is neither a direct option nor a command key, it splits the key into key path (everything up to the last "/") and key name (everything after the last "/"). Then, pgfkeys looks in the special /handlers/ subtree for a key called key name. This is then passed both the current path and the value given. For example,

```
\pgfkeys{/path/cmd key/.code = {(value=#1)}}
```

is a handled key with key name .code and key path /path/cmd key because

1. there is no direct key /path/cmd key/.code;
2. there is no command option by this name;
3. there *is* a command key /handlers/.code.

The predefined handler .code creates a new command key named according to the current key's path (in our case, /path/cmd key).

So, key handlers take a key path and a value as input and perform some kind of action with it. They can define new key types (for example storage keys, Boolean keys or choice keys as we will see in the next section), they can check whether a key is defined, they can change default paths and more. Much of the strength of the pgfkeys package comes from its key handlers.

8.4 Predefined key handlers

pgfkeys provides many predefined key handlers, most of which are used to define more or less special command keys. Here are some common key handlers:

[1] Again, pgfkeys uses a slightly different terminology. Its handled keys are not to be mistaken with the "handlers" defined by \define@key; those are called "command keys" in pgfkeys.

`.cd` A "change directory" command:

```
\pgfkeys{/path/.cd,A=a,B=b}
```

sets the default path to `/path` and will thus set `/path/A=a` and `/path/B=b`. We will later see that the command `\pgfqkeys` also changes the default path, thus

```
\pgfqkeys{/path}{A=a,B=b}
```

will also set `/path/A=a` and `/path/B=b`.

`.default={`⟨*value*⟩`}` Determines a value to be used if no "=" sign is given:

```
\pgfkeys{/path/A/.default=true}
\pgfkeys{/path/A}
```

is the same as if we had written

```
\pgfkeys{/path/A=true}
```

`.code={`⟨*code*⟩`}` Defines a new command key which expands to the value of `.code`. The resulting command key takes one argument.

`.is if={`⟨*TeX-Boolean*⟩`}` Creates a new Boolean key which sets a TeX Boolean to either true or false:

```
\newif\ifcoloured
\pgfkeys{
  /path/coloured/.is if = coloured
}
% set \colouredtrue:
\pgfkeys{/path/coloured=true}
% set \colouredfalse:
\pgfkeys{/path/coloured=false}
```

An error message is raised if the supplied value is neither `true` nor `false`. pgfkeys does not call `\newif` automatically, and the leading "`if`" must not be included in the argument of `.is if`, *i.e.* `coloured/.is if=ifcoloured` would be wrong.

`.is choice` Creates a new choice key, with the available choices given as subkeys of the current one:

```
\pgfkeys{
  /path/op/.is choice,
  /path/op/plus/.code={+},
  /path/op/minus/.code={-},
  /path/op/nop/.code={nothing}
}
% invokes /path/op/plus
\pgfkeys{/path/op=plus}
```

An error results if the user gives an unknown choice.

`.store in={`⟨`\macro`⟩`}` Defines a command key that simply stores its value into a macro:

```
\pgfkeys{/path/key/.store in=
  \keyvalue}
\pgfkeys{/path/key=my value}
Result is '\keyvalue'
```

Expands to "`Result is 'my value'`". Such a key is very similar to a *direct key*, see above.

`.style` Creates a new *style* key, which contains a list of other options. Whenever a style key is set, it sets all of its options:

```
\pgfkeys{
  /text/readable/.style=
    {font=large,color=pink},
  /text/unreadable/.style=
    {font=small,color=black}
}
\pgfkeys{/text/readable}
```

will set the additional options `/text/font=large` and `/text/color=pink` (using the default path since they have no full path).

`.append style` Appends more options to an already existing style key. Given the example above,

```
\pgfkeys{
  /text/readable/.append style=
    {underlined=true}}
```

has the same effect as writing

```
\pgfkeys{/text/readable/.style=
  {font=large,color=pink,
    underlined=true}}
```

Since style keys can be defined and changed easily, they provide much flexibility for package users.

8.5 pgfkeys in action — an example

We will now realise our example LaTeX package of Section 7 with pgfkeys. We use the same option names and the same user interface, with one exception: pgfkeys does not support LaTeX package options (although see Section 8.6). Any configuration has to be done with `\xmphsetup`.

We do not need to change our implementation for `\xmph` and we can keep its helper macros `\xmph@bold`, `\xmph@emph` and `\xmph@colourtext` as well. We only need to change the option declaration, which is shown in the following listing.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{xmph}
  [2009/03/17 v1.0 Extended emph]
\RequirePackage{color,pgfkeys}
\newif\ifxmph@useitalic
\newif\ifxmph@usebold
\newif\ifxmph@usecolour
\pgfkeys{
  /xmph/.cd,
  useitalic/.is if = xmph@useitalic,
  usebold/.is if = xmph@usebold,
  usecolour/.is if = xmph@usecolour,
  usecolor/.is if = xmph@usecolour,
  useitalic/.default = true,
```

```
  usebold/.default = true,
  usecolour/.default = true,
  usecolor/.style = {usecolour=#1},
  colour/.store in = \xmph@colour,
  color/.style = {colour=#1},
  inactive/.code = {%
    \let\xmph\emph
    \PackageInfo{xmph}
      {Package inactive}%
    }
}
\pgfkeys{
  /xmph/.cd,
  useitalic,
  colour = red
}
\newcommand*{\xmphsetup}{%
  \pgfqkeys{/xmph}%
}
\AtBeginDocument{
  \pgfkeys{
    /xmph/inactive/.code = {%
      \PackageInfo{xmph}{%
        Option 'inactive' only
        available in preamble
      }%
    }
  }
}
```

The command \pgfqkeys occurring in the last listing is a variant of \pgfkeys which sets the default path directly, without a .cd statement. The command

```
\pgfqkeys{/xmph}{
  colored = false,
  bold    = true
}
```

thus uses /xmph as its default path.

8.6 pgfkeys for LaTeX package options

The pgfkeys package does not include any native functionality for processing LaTeX package and class options. However, the pgfopts package (Wright, 2008) adds this ability, using a modified copy of the functionality in kvoptions.

The pgfopts package provides only a single user macro, \ProcessPgfOptions. Keys are created using the pgfkeys interface discussed above, and can then be used as package (or class) options using the \ProcessPgfOptions macro. The requirement to have *no* spaces in the key names for this to work remains exactly the same as for xkeyval or kvoptions processing of options.

9 Conclusions

There are a number of methods for the author wanting to make a start using key–value input. The pgfkeys package has much to recommend it. The interface has been well designed, and it is very strong in handling a wide range of situations (well illustrated in the user documentation). For large-scale projects in particular, the tree concept makes option management much easier. By loading pgfopts, LaTeX option processing is also possible with pgfkeys.

For users who wish to handle LaTeX package options using key–value input, most authors will want to load either kvoptions or xkeyval, rather than coding the option handler directly. Both handle the core issue of providing key–value package options well. Each packages has some advantages, depending on the job at hand.

xkeyval provides a rich set of macros for defining almost every possible type of key. The additional graduation of keys made available by the variable prefix is welcome. The package has a very large number of features which have not been discussed here. However, the package has been criticised for modifying keyval internals. More importantly for many, it suffers from the very problem of complex optional arguments that the key–value method is supposed to avoid.

On the other hand, kvoptions provides a smaller, but more focussed, set of additional key types. The input syntax is much less complex than that of xkeyval, and the provision of \SetupKeyvalOptions is particularly welcome. Using the kvoptions method does make it more likely that ambitious package authors will have to become familiar creating customised functions with \define@key. However, the clearer syntax make kvoptions a better choice for rapidly making progress with using key–value input.

10 Acknowledgments

Thanks to Didier Verna and Morten Høgholm for helpful suggestions when drafting this manuscript, and Will Robertson for the example of the keyval *versus* xkeyval problem.

References

Adriaens, Hendri. "The xkeyval package". Available from CTAN, macros/latex/contrib/xkeyval, 2008.

Adriaens, Hendri, and U. Kern. "xkeyval — new developments and mechanisms in key processing". *TUGboat* **25**(2), 194–199, 2005.

Carlisle, David. "The keyval package". Part of the graphics bundle, available from CTAN, macros/latex/required/tools, 1999.

Oberdiek, Heiko. "The kvoptions package". Part of the oberdiek bundle, available from CTAN, macros/latex/contrib/oberdiek, 2009a.

Oberdiek, Heiko. "The kvsetkeys package". Part of the oberdiek bundle, available from CTAN, macros/latex/contrib/oberdiek, 2009b.

Tantau, Till. "pgfkeys". Part of the TikZ and PGF bundle, available from CTAN, graphics/pgf, 2008.

Wright, Joseph. "pgfopts — LaTeX package options with pgfkeys". Available from CTAN, macros/latex/contrib/pgfopts, 2008.

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

◇ Christian Feuersänger
Institute for Numerical Simulation
Wegelerstraße 6
53115 Bonn
Germany
ludewich (at) users dot sourceforge dot net