Summary

The ltxkeys package provides facilities for creating and managing keys in the manner of the keyval and xkeyval packages, but it is intended to be more robust and faster than these earlier packages. Yet it comes with many new functions.

The ltxkeys Package☆,★

A robust key parser

Ahmed Musa[1,2]

20th November 2011

Contents

☆ The package is available at http://mirror.ctan.org/macros/latex/contrib/ltxkeys/.
★ This user manual corresponds to version 0.0.2 of the package.

[1] University of Central Lancashire, Preston, UK.
[2] Email address for all ltxkeys package related matters: amusa22@gmail.com.

1 Introduction

THE LTXKEYS PACKAGE provides facilities for creating and managing keys in the manner of the keyval and xkeyval packages, but it is intended to be more robust and faster than these earlier packages. Its robustness emanates from, inter alia, its ability to preserve braces in key values throughout parsing. The need to preserve braces in key values without expecting the user to double braces emerges often in parsing keys. This is the case in, e. g., the xwatermark package, but consider also the possibility of passing all the following options to a package at once, where 'layout' is a package or class option or key[⋆1]:

Braced options

```
\pkgoptions{%
  opt1=val1,opt2=val2,
  layout={left=3cm,right=3cm,top=2.5cm,bottom=2.5cm,include=true}
}
```

The ltxkeys package is faster[⋆2] than the xkeyval package mainly because it avoids character-wise parsing of key values (which is called 'selective sanitization' by the xkeyval package). Moreover,

[⋆1] It should be noted that if a value of the demonstrative option layout is expandable, then the option can't be passed by \documentclass without preloading a robust options parser like kvoptions-patch, xkvltxp, catoptions, or ltxkeys package. The ltxkeys package, unlike the xkeyval package, can be loaded before \documentclass.

[⋆2] Because of the multitude of functions provided by the ltxkeys package, it may actually slow down when executing some tasks, depending on the task at hand. The package option tracingkeys, for example, does slow down

it is faster to normalize a comma-separated or ⟨key⟩=⟨value⟩ list than trim leading and trailing spaces of each element of the list (as the xkeyval package does), since not all the elements of the list will normally have leading and trailing spaces. In fact, the chances are that only less than 50 percent of the elements of the list will have such spaces. As another example of optimization, anyone familiar with the implementation of the xkeyval package would have noticed that the macro \XKV@srstate, which (in order to allow \setkeys to be re-entrant) pushes and pops the states of some important functions in the package, loops over all the functions both when pushing and popping. In the ltxkeys package, pushing and popping functions together involve looping over the functions only once. And, unlike in the xkeyval package, higher order functions are undefined as soon as they are no longer needed, to avoid clogging up the stack. No additional looping is required for this.

In setting keys, the ltxkeys package loops over not only families, as in the xkeyval package, but also over key prefixes. The same strategy applies when the ltxkeys package tries to establish if a key is defined or not.

While some user interfaces of the ltxkeys package are similar to those of the xkeyval package, there are important differences in several areas of syntax, semantics, and internal implementation. The ltxkeys package also provides additional facilities (beyond the xkeyval package) for defining and managing keys. Several types of keys (including ordinary keys, command keys, style keys, choice keys, boolean and biboolean keys) can be efficiently created and managed. In the ltxkeys package, the notions of 'pre-setting' and 'post-setting' keys are similar to those of the xkeyval package. But the ltxkeys package introduces additional concepts in this respect: 'initialized' and 'launched' keys. The latter are special preset keys. The pointer system of the xkeyval package, which was available only at key-setting time, is now also available at key definition time. One more type of pointer (\needvalue) has been introduced to require users of 'need-value keys' to supply values for those keys.

Rather than simply issue an error for undefined keys when setting keys, the ltxkeys package provides the 'undefined keys' and 'undefined options' handlers, which are user-customizable. Other new concepts include 'definable keys', 'cross-family keys', 'option keys', 'non-option keys', 'handled keys', 'pathkeys', 'key commands', 'key environments', accessing the saved value of a key outside \setkeys or similar commands, and declaring multiple keys (of all genre) using only one command.

It not advisable to alias the commands of the xkeyval package to the commands of the ltxkeys package. There are many existing packages that rely on the xkeyval package and aliasing commands that are used by other packages can cause confusion[⋆3].

1.1 Motivation

What are the *raison d'etre* and origins of the ltxkeys package? Well, I decided to write this package as I grabbled with some practical problems of key parsing while developing version 1.5.0 of the xwatermark package. The tasks proved more challenging than I had initially thought and, despite its commendable and widely deployed features, I found the xkeyval package inadequate in some respects. As mentioned earlier, all the functions of the ltxkeys package can be employed for general key management in LaTeX beyond the xwatermark package. Indeed the ltxkeys package can be used as a more robust replacement for the xkeyval package, of course with modifications

processing. And automatically initiating keys after definition, as done by the commands \ltxkeys@definekeys and \ltxkeys@declarekeys, also affects processing speed; so does 'launching keys,' which first presets absent keys with their default values before setting the current keys (i. e., keys whose values are provided by the user at the moment of setting keys that belong to a family). Then, as in the xkeyval package, there are the commands for presetting and post-setting keys.

⋆3 A user of version 0.0.1 of the package had sought to do this.

of names and some syntaxes. The xkeyval package has been frozen since August 2008.

2 Package options

The package options are listed in Table 1. The package options can be passed via \documentclass, \RequirePackage or \usepackage as follows:

Example: Package options

```
\documentclass[tracingkeys,keyparser={|},pathkeys]{article}
or
\usepackage[tracingkeys,keyparser={|}]{ltxkeys}
```

They can also be passed via the command \ltxkeys@options:

New macro: \ltxkeys@options

```
\ltxkeys@options{tracingkeys=false,keyparser={;}}
```

Table 1: Package options

Option	Default	Meaning
tracingkeys	false	The global boolean switch that determines if information should be logged in the transcript for some tasks in the package.[See note 1.1]
keyparser	;	The list parser used by some internal loops in defining keys.[1.2]
keydepthlimit	4	This is used to guard against erroneous infinite re-entrance of the package's key-setting commands. The default value of 4 means that neither of these commands can ordinarily be nested beyond level 4.[1.3]
reservenopath	false	The 'path' (or roots or bases) of a key is the combination of key prefix, key family and macro prefix, but when dealing with 'pathkeys' (see section 15) the term excludes the macro prefix. These can be reserved and unreserved by any user by the tools of section 9. Subsequent users can, at their own risk, override all previously reserved paths by enabling the package's boolean option reservenopath.
allowemptypath	false	Allow the use of empty key prefix and family. This isn't advisable but some pre-existing packages might have used empty key prefixes and families.[1.4]
pathkeys	false	Load the pathkeys package (see section 15).

Table 1 notes

[1.1] The speed of compilation may be affected by this option, but it is recommended at the pre-production stages of developing keys. The option provide some trace functionality and enables the user to, among other things, follow the progress of the LaTeX run and to see if a key has been defined and/or set/executed more than once in the current run. The starred (⋆) variants of the commands \ltxkeys@definekeys and \ltxkeys@declarekeys will always flag an error if a key is being defined twice, irrespective of the state of the package option tracingkeys. The \ltxkeys@xxxkey variants (unlike the \ltxkeys@newxxxkey variants) of key-defining commands don't have this facility, and it may be desirable to know if and when an existing key is being redefined.

[1.2] Wherever the semicolon ';' is indicated as a list parser in this guide, it can be replaced by any user-specified one character parser via the package option keyparser. To avoid confusing the user-supplied parser with internal parsers, it is advisable to enclose the chosen character in curly braces. The braces

will be stripped off internally. Please note that some of the characters that may be passed as a list parser may indeed be active; be careful to make them innocent before using them as a list/key parser. My advice is that the user sticks with the semicolon ';' as the key parser: the chances of it being made active by any package is minimal. If you have the chosen parser as literals in the callbacks of your keys, they have to be enclosed in curly braces.

[1.3] The key-setting commands are \ltxkeys@setkeys, \ltxkeys@setrmkeys and \ltxkeys@setaliaskey. If you must nest these commands beyond level 4, you have to raise the keydepthlimit as a package option. The option keystacklimit is an alias for keydepthlimit.

[1.4] The use of an empty prefix will normally result from explicitly declaring the prefix as [], rather than leaving it undeclared. Undeclared prefixes assume the default value of KV. An empty family will result from submitting the family as empty balanced curly braces {}. If keys lack prefix and/or family, there is a strong risk of confusing key macros/functions. For example, without a prefix and/or family, a key named width will have a key macro defined as \width, which portents sufficient danger.

3 Defining keys

3.1 Defining only definable keys

If the package option tracingkeys is enabled (i. e., turned true), the user can see in the transcript file the existing keys that he has redefined with the \ltxkeys@xxxkey variants of the key-defining commands, which redefine existing keys without any default warning or error. The log file messages being referred to here will be highlighted with the warning sign (!!). This is always desirable in the preproduction stages of your project. However, instead of looking for these warning messages in the log file, the user can use the \ltxkeys@newxxxkey variants of the key-defining commands to bar himself from redefining existing keys.

Subsequently we will mention the \ltxkeys@newxxxkey variants of key-defining commands without necessarily explaining what they mean, since their meaning is henceforth clear.

In the following, syntactic quantities in square brackets (e. g., [yyy]) and those in parenthesis (e. g., (yyy)) are optional arguments.

3.2 Ordinary keys

New macros: \ltxkeys@ordkey, \ltxkeys@newordkey

```
\ltxkeys@ordkey[⟨pref⟩]{⟨fam⟩}{⟨key⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@newordkey[⟨pref⟩]{⟨fam⟩}{⟨key⟩}[⟨dft⟩]{⟨cbk⟩}
```

These define a macro of the form \⟨pref⟩@⟨fam⟩@⟨key⟩ of one parameter that holds the key function/callback ⟨cbk⟩. The default value for the 'key prefix' ⟨pref⟩ is always KV, as in the xkeyval package. When ⟨key⟩ is used in a \ltxkeys@setkeys command (see section 4) containing ⟨key⟩=⟨value⟩, the macro \⟨pref⟩@⟨fam⟩@⟨key⟩ takes the value as its argument and is then executed. The given argument or key value can be accessed in the key's callback ⟨cbk⟩ by using #1 inside the function. The optional default value ⟨dft⟩, if available, will be used by \⟨pref⟩@⟨fam⟩@⟨key⟩ when the user hasn't provided a value for the key at \ltxkeys@setkeys. If ⟨dft⟩ was absent at key definition and the key user hasn't provided a value for the key, an error message is flagged[⋆4].

Run the following example and do \show\cmdb and \show\cmdd:

[⋆4] The commands \ltxkeys@key and \ltxkeys@newkey aren't user commands.

Example: \ltxkeys@ordkey

```
\ltxkeys@ordkey[KV]{fam}{keya}[\def\cmda#1{aa#1}]{\def\cmdb##1{#1bb##1}}
\ltxkeys@ordkey[KV]{fam}{keyb}[\def\cmdc##1{cc##1}]{\def\cmdd##1{#1dd##1}}
\ltxkeys@setkeys[KV]{fam}{keya,keyb}
```

3.2.1 Ordinary keys that share the same attributes

The commands \ltxkeys@ordkey and \ltxkeys@newordkey can be used to introduce ordinary keys ⟨keys⟩ that share the same path[★5] (key prefix, key family, and macro prefix) and callback ⟨cbk⟩. All that is needed is to replace ⟨key⟩ in these commands with the comma-separated list ⟨keys⟩. Because some users might prefer to see these commands in their plural forms when defining several keys with the same callback, we have provided the following aliases. The internal coding remains the same and no efficiency has been lost in generalization.

New macros: \ltxkeys@ordkeys, \ltxkeys@newordkeys

```
\ltxkeys@ordkeys[⟨pref⟩]{⟨fam⟩}{⟨keys⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@newordkeys[⟨pref⟩]{⟨fam⟩}{⟨keys⟩}[⟨dft⟩]{⟨cbk⟩}
```

3.3 Command keys

New macros: \ltxkeys@cmdkey, \ltxkeys@newcmdkey

```
\ltxkeys@cmdkey[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨key⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@newcmdkey[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨key⟩}[⟨dft⟩]{⟨cbk⟩}
```

Here, the optional quantity ⟨mp⟩ is the 'macro prefix.' If ⟨mp⟩ is given, the command \⟨mp⟩⟨key⟩ will hold the current user input at key setting time; otherwise (i. e., if ⟨mp⟩ is absent) the user input will be available in the macro \cmd⟨pref⟩@⟨fam⟩@⟨key⟩. The command \⟨pref⟩@⟨fam⟩@⟨key⟩ is the 'key macro' and will hold the callback ⟨cbk⟩. This type of key is traditionally called 'command key' (a name that most likely emanated from the xkeyval package) because it gives rise to the macro \⟨mp⟩⟨key⟩, but in the ltxkeys package even boolean, style and choice keys are associated with this type of macro.

3.3.1 Command keys that share the same attributes

The commands \ltxkeys@cmdkey and \ltxkeys@newcmdkey can be used to introduce command keys ⟨keys⟩ that share the same path or bases (key prefix, key family, and macro prefix) and callback ⟨cbk⟩. Simply replace ⟨key⟩ in these commands with the comma-separated list ⟨keys⟩. Some users might prefer to see these commands in their plural forms when defining several keys with the same callback. We have therefore provided the following aliases:

New macros: \ltxkeys@cmdkeys, \ltxkeys@newcmdkeys

```
\ltxkeys@cmdkeys[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨keys⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@newcmdkeys[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨keys⟩}[⟨dft⟩]{⟨cbk⟩}
```

[★5] The key path is also called the key bases.

3.4 Style keys

Style keys are keys with dependants (i. e., keys that are processed when the master is set). They have the following syntaxes:

New macros: \ltxkeys@stylekey, \ltxkeys@newstylekey

```
\ltxkeys@stylekey[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨key⟩}[⟨dft⟩](⟨deps⟩){⟨cbk⟩}
\ltxkeys@stylekey*[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨key⟩}[⟨dft⟩](⟨deps⟩){⟨cbk⟩}
\ltxkeys@newstylekey[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨key⟩}[⟨dft⟩](⟨deps⟩){⟨cbk⟩}
\ltxkeys@newstylekey*[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨key⟩}[⟨dft⟩](⟨deps⟩){⟨cbk⟩}
```

The dependants ⟨deps⟩ have the syntax:

Dependant keys syntax

```
(
  ⟨keytype⟩/⟨keyname⟩/⟨dft⟩/⟨cbk⟩;
  another set of dependant; etc.
)
```

The default value ⟨dft⟩ and the callback ⟨cbk⟩ can be absent in the syntax of style keys. ⟨keytype⟩ can be 'ord' (ordinary key), 'cmd' (command key), 'bool' (boolean key), or 'choice' (choice key).

Dependant keys always share the same key prefix ⟨pref⟩, family ⟨fam⟩, and macro prefix ⟨mp⟩ with the parent key.

If ⟨mp⟩ is given, the command \⟨mp⟩⟨key⟩ will hold the current user input for the parent key; otherwise the user input will be available in \style⟨pref⟩@⟨fam⟩@⟨key⟩. The macro \⟨pref⟩@⟨fam⟩@⟨key⟩ will always hold the callback ⟨cbk⟩.

If the starred (⋆) variant is used, all undefined dependants will be defined and set on the fly as the parent is being set. If the starred (⋆) variant isn't used and undefined dependants occur, then an error message will be flagged at the time the parent is being set.

Most of the time it is possible to access the parent key's current value with \parentval. Within ⟨dft⟩ and ⟨cbk⟩ of ⟨deps⟩, it is possible to refer to the parent key's callback with its full macro name (i. e., \⟨pref⟩@⟨fam⟩@⟨key⟩). \parentval is always available for use as the default value of dependant keys, but it may be lost in the callbacks of dependant keys, because a dependant key, once defined, may be set independent of, and long after, the parent key has been executed. It is therefore more reliable to refer to the macro \⟨pref⟩@⟨fam⟩@⟨key⟩@value, which is recorded for only the parent key of style keys and which holds the current user input for the parent key. The macro \⟨pref⟩@⟨fam⟩@⟨key⟩@value is recorded only if it appears at least once in the attributes of dependant keys. The macro \⟨pref⟩@⟨fam⟩@⟨key⟩@value has a more unique name than \⟨mp⟩⟨key⟩ but they always contain the same value of a style key. As mentioned above, if ⟨mp⟩ is not given, the user input for a style key will be available in the macro \style⟨pref⟩@⟨fam⟩@⟨key⟩, instead of \⟨mp⟩⟨key⟩.

Note 3.1 '#1' in the callback of parent key refers to the current value of the parent key, while '#1' in the callback of any dependant key refers to the current value of that dependant key. Here is an example that defines and sets all undefined dependants on the fly:

Examples: \ltxkeys@stylekey

```
\ltxkeys@stylekey*[KV]{fam}[mp@]{keya}[{left}](%
  % '#1' here refers to the value of the DEPENDANT key
  % at the time it is being set. Use \parentkey and \parentval
```

```
    % here to access the parent key name and its current value:
    ord/keyb/{right}/\def\y##1{#1##1};
    %  The default of keyc is the current value of parent (keya):
    cmd/keyc/\parentval;
    % Because \KV@fam@keya@value appears below, it will be saved
    % when the parent key keya is being set, otherwise it would be
    % unavailable:
    bool/keyd/true/\ifmp@keyd\edef\x##1{##1\KV@fam@keya@value}\fi
  ){%
    % '#1' here refers to the value of the PARENT key at the time
    % it is being set:
    \def\x##1{##1xx#1xx}%
    % Check the value of parent key:
    \ltxkeys@checkchoice[,](\userinput\order){#1}{left,right,center}{}{%
      \@latex@error{Invalid input '#1'}\@ehd
    }%
  }
```

In this example, `\userinput` corresponds to `#1`, and `\order` is the numerical order of the user input in the nominations `{left | right | center}`. More about the commands `\ltxkeys@checkchoice` and `\CheckUserInput` can be found in subsection 16.2.

You can try setting `keya` as follows to see what happens to keys `keyb`, `keyc` and `keyd`:

Example: `\ltxkeys@setkeys`

```
\ltxkeys@setkeys[KV]{fam}{keya=right}
```

The following will flag an error because `{right}` isn't in the list of nominations `{left | right | center}`:

Example: `\ltxkeys@setkeys`

```
\ltxkeys@setkeys[KV]{fam}{keya={right}}
```

The braces in the key values above are just to exemplify the fact that braces in key values are preserved throughout key parsing. As mentioned earlier, this is essential for some packages and class files.

3.4.1 Style keys that share the same attributes

The commands `\ltxkeys@stylekey` and `\ltxkeys@newstylekey` can be used to introduce style keys ⟨keys⟩ that share the same path or bases (key prefix, key family, and macro prefix) and callback ⟨cbk⟩. Just replace ⟨key⟩ in these commands with the comma-separated list ⟨keys⟩. However, some users might prefer to see these commands in their plural forms when defining several keys with the same callback. Hence, we also provide the following aliases:

New macros: `\ltxkeys@stylekeys, \ltxkeys@newstylekeys`

```
\ltxkeys@stylekeys[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨keys⟩}[⟨dft⟩](⟨deps⟩){⟨cbk⟩}
\ltxkeys@stylekeys*[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨keys⟩}[⟨dft⟩](⟨deps⟩){⟨cbk⟩}
\ltxkeys@newstylekeys[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨keys⟩}[⟨dft⟩](⟨deps⟩){⟨cbk⟩}
\ltxkeys@newstylekeys*[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨keys⟩}[⟨dft⟩](⟨deps⟩){⟨cbk⟩}
```

3.5 Boolean keys

New macros: \ltxkeys@boolkey, \ltxkeys@newboolkey

```
\ltxkeys@boolkey[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨key⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@boolkey+[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨key⟩}[⟨dft⟩]{⟨cbk⟩}{⟨fn⟩}
\ltxkeys@newboolkey[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨key⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@newboolkey+[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨key⟩}[⟨dft⟩]{⟨cbk⟩}{⟨fn⟩}
```

In these commands, if ⟨mp⟩ is given, the command \⟨mp⟩⟨key⟩ will hold the current user input for the key at key setting time; otherwise the user input will be available in \bool⟨pref⟩@⟨fam⟩@⟨key⟩[★6]. If ⟨mp⟩ is specified, a boolean of the form \if⟨mp⟩⟨key⟩ will be created at key definition, which will be set by \ltxkeys@setkeys according to the user input. If ⟨mp⟩ is not specified, a boolean of the form \ifbool⟨pref⟩@⟨fam⟩@⟨key⟩ will instead be created.

The user input for boolean keys must be in the set {true | false}. The callback ⟨cbk⟩ is held in the command \⟨pref⟩@⟨fam⟩@⟨key⟩, which is executed if the user input is valid.

The plus (+) variant of \ltxkeys@boolkey and \ltxkeys@newboolkey will execute ⟨fn⟩ in place of ⟨cbk⟩ if the user input isn't in {true | false}; the plain form will issue an error in this case.

3.5.1 Boolean keys that share the same attributes

The commands \ltxkeys@boolkey and \ltxkeys@newboolkey can be used to introduce boolean keys ⟨keys⟩ that share the same path or bases (key prefix, key family, and macro prefix) and callback ⟨cbk⟩. Just replace ⟨key⟩ in these commands with the comma-separated list ⟨keys⟩. Because some users might prefer to see these commands in their plural forms when defining several keys with the same callback, we have provided the following aliases:

New macros: \ltxkeys@boolkeys, \ltxkeys@newboolkeys

```
\ltxkeys@boolkeys[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨keys⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@boolkeys+[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨keys⟩}[⟨dft⟩]{⟨cbk⟩}{⟨fn⟩}
\ltxkeys@newboolkeys[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨keys⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@newboolkeys+[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨keys⟩}[⟨dft⟩]{⟨cbk⟩}{⟨fn⟩}
```

3.5.2 Biboolean keys

New macros: \ltxkeys@biboolkeys, \ltxkeys@newbiboolkeys

```
\ltxkeys@biboolkeys[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨bl1⟩,⟨bl2⟩}[⟨dft⟩]{⟨cbk1⟩}{⟨cbk2⟩}
\ltxkeys@biboolkeys+[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨bl1⟩,⟨bl2⟩}[⟨dft⟩]{⟨cbk1⟩}{⟨cbk2⟩}{⟨fn⟩}
\ltxkeys@newbiboolkeys[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨bl1⟩,⟨bl2⟩}[⟨dft⟩]{⟨cbk1⟩}{⟨cbk2⟩}
\ltxkeys@newbiboolkeys+
  [⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨bl1⟩,⟨bl2⟩}[⟨dft⟩]{⟨cbk1⟩}{⟨cbk2⟩}{⟨fn⟩}
```

Biboolean keys always assume opposite states: when one is true, the other is automatically toggled to false; and vice versa. Think of the options draft and final in a document class, but note that traditional document classes don't currently use biboolean keys. The callback ⟨cbk1⟩ belongs to the boolean key ⟨bl1⟩, while ⟨cbk2⟩ is of ⟨bl2⟩.

★6 This differs from the system in the xkeyval package.

The plus (+) variant of \ltxkeys@biboolkeys will execute ⟨fn⟩ in place of ⟨cbk1⟩ or ⟨cbk2⟩ if the input is not in {true | false}; the plain form will issue an error in this case.

Biboolean keys have equal symmetry (i. e., they can call each other with equal propensity) and they won't bomb out in an infinite reentrance. They normally would know if and when they call each other, or if they're being called by some other keys.

Examples: \ltxkeys@biboolkeys

```
\ltxkeys@biboolkeys+[KV]{fam}[mp@]{keya,keyb}[true]{%
  \ifmp@keya\def\x##1{##1x#1x##1}\fi
}{%
  \ifmp@keyb\def\y##1{##1y#1y##1}\fi
}{%
  \@latex@error{Invalid value `\string#1' for keya or keyb}\@ehc
}
```

3.6 Choice keys

The choice keys of the ltxkeys package differ from those of the xkeyval package in at least two respects; namely, the presence of the macro prefix for choice keys in the ltxkeys package and the introduction of the optional '!' prefix.

New macros: \ltxkeys@choicekey, \ltxkeys@newchoicekey

```
\ltxkeys@choicekey[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨key⟩}[⟨bin⟩]{⟨alt⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@choicekey*[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨key⟩}[⟨bin⟩]{⟨alt⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@choicekey*+[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨key⟩}[⟨bin⟩]{⟨alt⟩}[⟨dft⟩]{⟨cbk⟩}{⟨fn⟩}
\ltxkeys@choicekey*+![⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨key⟩}[⟨bin⟩]{⟨alt⟩}[⟨dft⟩]{⟨cbk⟩}{⟨fn⟩}

\ltxkeys@newchoicekey[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨key⟩}[⟨bin⟩]{⟨alt⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@newchoicekey*[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨key⟩}[⟨bin⟩]{⟨alt⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@newchoicekey*+
  [⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨key⟩}[⟨bin⟩]{⟨alt⟩}[⟨dft⟩]{⟨cbk⟩}{⟨fn⟩}
\ltxkeys@newchoicekey*+!
  [⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨key⟩}[⟨bin⟩]{⟨alt⟩}[⟨dft⟩]{⟨cbk⟩}{⟨fn⟩}
```

Choice keys check the user input against the nominations ⟨alt⟩ suggested by the author of a key. The comma-separated list ⟨alt⟩ is the list of admissible values of the key. The starred (*) variant will convert user input to lowercase before checking it against the list of nominations in ⟨alt⟩. In all the above variants, if the input is valid, then the callback ⟨cbk⟩ will be executed. If the user input isn't valid, the non-plus variants will flag an error, while the plus (+) variants will execute ⟨fn⟩. The ! variants will fully expand the user input before checking it against the nominations in ⟨alt⟩. The ! variant arises from the fact that sometimes macros are passed as the values of choice keys. If ⟨mp⟩ is absent, then \ltxkeys@choicekey uses \chc⟨pref⟩@⟨fam⟩@⟨key⟩ to hold the user input.

When ⟨alt⟩ has no literal form '/.code' or forward slash '/' in it, then it is expected to be of the familiar xkeyval package syntax:

Syntax of 'nominations' for choice keys

```
{choice1,choice2,etc.}
```

If ⟨alt⟩ has '/.code' or '/' in it, then it is expected to have one of the following syntaxes:

Syntaxes of 'nominations' for choice keys

```
{%
  choice1/.code=callback1⟨keyparser⟩
  choice2/.code=callback2⟨keyparser⟩
  etc.
}

{%
  choice1/callback1⟨keyparser⟩
  choice2/callback2⟨keyparser⟩
  etc.
}
```

If the parser is semicolon ';', then we would have

Syntaxes of 'nominations' for choice keys

```
{choice1/.code=callback1; choice2/.code=callback2; etc.}

{choice1/callback1; choice2/callback2; etc.}
```

This means that if you have '/.code' or '/' in any of the callbacks, it has to be enclosed in curly braces. Please recall that the default value of ⟨keyparser⟩ is semicolon ';'. keyparser is a package option. This syntax also implies that if you have the ⟨keyparser⟩ in ⟨defn⟩, it has to be wrapped in curly braces. The ⟨keyparser⟩ in this syntax could also be comma ','.

Note 3.2 Here is the rule for parsing the ⟨alt⟩ list. First the package checks if the declared key parser (i. e., ⟨keyparser⟩) is in the ⟨alt⟩ list. If the parser exists in ⟨alt⟩, then the list is parsed using this parser. Otherwise the list is parsed using comma as the parser. Moreover, the package checks if '.code' separates ⟨choice⟩ from the callback ⟨cbk⟩. If no '.code' is found, then '/' is assumed to be the separator. But note that when there is no ⟨cbk⟩, then neither '.code' nor '/' is necessary.

It is possible to refer to the current value of ⟨key⟩ as #1 in ⟨alt⟩.

The starred (⋆) variant of \ltxkeys@choicekey will convert the user input to lowercase before checking ⟨alt⟩ and executing the callbacks. The plus (+) variant will execute ⟨fn⟩ in place of ⟨cbk⟩ if the user input isn't in ⟨alt⟩.

⟨bin⟩ has, e. g., the syntax [\userinput\order], where \userinput will hold the user input (in lowercase if the starred (⋆) variant of \ltxkeys@choicekey is called), and \order will hold the serial number of the value in the list of nominations ⟨alt⟩, starting from 0. If the input isn't valid, \userinput will still hold the user input, but \order will be −1.

Examples: \ltxkeys@choicekey nominations

```
\ltxkeys@choicekey[KV]{fam}{keya}{%
  % There are no callbacks for these simple nominations:
  center,right,left,justified
}[center]{%  <- default value
  \def\x##1##2{==##1++#1++##2==}%
}

\ltxkeys@choicekey*+[KV]{fam}[mp@]{keya}[\userinput\order]{%
  center,right,left,justified
}[center]{%
```

```
  \def\x##1##2{==##1++#1++##2==}%
}{%
  \@latex@error{Inadmissible value `\detokenize{#1}' for keya}\@ehc
}

\ltxkeys@choicekey*+[KV]{fam}[mp@]{keyb}[\userinput\order]{%
  % There are callbacks for these nominations:
  land/.code=\def\x##1{*##1*#1*##1};
  air/.code=\edef\z{\expandcsonce\ltxkeys@tval};
  sea/.code=\edef\myinput{\cpttrimspaces{#1}};
  space/.code=\letcsntocs{#1@earth}\relax
}[center]{%
  \def\z##1##2{==##1++#1++##2==}%
}{%
  \@latex@error{Inadmissible value `\detokenize{#1}' for keya}\@ehc
}

\ltxkeys@choicekey[KV]{fam}[mp@]{keyb}[\userinput\order]{%
  % The callbacks can also take the following form:
  center/\ltxkeys@cmdkey[KV]{fam}[mp@]{keyd}{\def\x####1{####1*##1*####1}},
  right/\let\align\flushright,
  left/\let\align\flushleft\edef\userinput{\cpttrimspaces{#1}},
  justified/\let\align\relax
}[center]{%
  \def\z##1##2{==##1++#1++##2==}%
}

\ltxkeys@choicekeys[KV]{fam}[mp@]{keya,\savevalue\needvalue{keyb}}%
[\val\order]{%
  center/\ltxkeys@cmdkey[KV]{fam}[mp@]{keyd}[\usevalue{keyb}]
    {\def\x####1{####1*##1*####1}},
  right/\def\y##1{##1++#1++##1},
  left/\edef\userinput{\cpttrimspaces{#1}},
  justified/\letcsntocs{#1@align}\relax
}[center]{%
  \def\z##1##2{==##1++#1++##2==}%
}
\ltxkeys@setkeys[KV]{fam}{keyb=center,keyd}
```

The representations \savevalue, \usevalue and \needvalue are pointers (see subsection 4.4).

3.6.1 Choice keys that share the same attributes

The commands \ltxkeys@choicekey and \ltxkeys@newchoicekey can be used to introduce choice keys ⟨keys⟩ that share the same path or bases (key prefix, key family, and macro prefix) and callback ⟨cbk⟩. All the user has to do is to replace ⟨key⟩ in these commands with the comma-separated list ⟨keys⟩. Some users might prefer to see these commands in their plural forms when defining several keys with the same attributes. We have therefore provided the following aliases without modifying the internal coding:

New macros: \ltxkeys@choicekeys, \ltxkeys@newchoicekeys

```
\ltxkeys@choicekeys[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨keys⟩}[⟨bin⟩]{⟨alt⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@choicekeys*[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨keys⟩}[⟨bin⟩]{⟨alt⟩}[⟨dft⟩]{⟨cbk⟩}
```

```
\ltxkeys@choicekeys*+
  [⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨keys⟩}[⟨bin⟩]{⟨alt⟩}[⟨dft⟩]{⟨cbk⟩}{⟨fn⟩}
\ltxkeys@choicekeys*+!
  [⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨keys⟩}[⟨bin⟩]{⟨alt⟩}[⟨dft⟩]{⟨cbk⟩}{⟨fn⟩}
\ltxkeys@newchoicekeys[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨keys⟩}[⟨bin⟩]{⟨alt⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@newchoicekeys*[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨keys⟩}[⟨bin⟩]{⟨alt⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@newchoicekeys*+
  [⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨keys⟩}[⟨bin⟩]{⟨alt⟩}[⟨dft⟩]{⟨cbk⟩}{⟨fn⟩}
\ltxkeys@newchoicekeys*+!
  [⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨keys⟩}[⟨bin⟩]{⟨alt⟩}[⟨dft⟩]{⟨cbk⟩}{⟨fn⟩}
```

3.7 Defining boolean and command keys with one command

In my personal experience, boolean and command keys have been the most widely used types of key in the context of xkeyval package. More than one boolean and command keys can be defined simultaneously by the following command:

New macro: \ltxkeys@definekeys

```
\ltxkeys@definekeys[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{%
  ⟨key⟩=⟨dft⟩/⟨cbk⟩;
  another set of key attributes; etc.
}
\ltxkeys@definekeys*[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{%
  ⟨key⟩=⟨dft⟩/⟨cbk⟩;
  another set of key attributes; etc.
}
```

The default value ⟨dft⟩ can be absent in the case of command keys, and the callback ⟨cbk⟩ can be absent for the two types of key. Boolean keys must, however, have default values {true | false}, to be distinguishable from command keys. The equality sign (=) that separates the key name from the default value can be replaced with forward slash (/). That is, the following syntax is also permitted:

New macro: \ltxkeys@definekeys

```
\ltxkeys@definekeys[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{%
  ⟨key⟩/⟨dft⟩/⟨cbk⟩;
  another set of key attributes; etc.
}
\ltxkeys@definekeys*[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{%
  ⟨key⟩/⟨dft⟩/⟨cbk⟩;
  another set of key attributes; etc.
}
```

You can use the command \CheckUserInput in ⟨cbk⟩ to indirectly introduce choice keys as command keys (see example below).

Ordinary keys and conventional choice keys can't be introduced directly by this command (use the command \ltxkeys@declarekeys instead).

The starred (⋆) variant of \ltxkeys@definekeys can be used to define non-existing boolean and command keys in the sense of \newcommand.

Note 3.3 Keys defined by \ltxkeys@definekeys are automatically set/initialized instantly, to

provide default values for immediate use. Boolean keys are preset with value 'false', so that they aren't turned 'true' prematurely.

Note 3.4 In \ltxkeys@definekeys and \ltxkeys@declarekeys every line is assumed to end with a comment sign. This is to be specially noted if a space is desired at the end of line. You can insert such a space with a comment sign, or, if appropriate, use \space.

Examples: \ltxkeys@definekeys

```
% The starred (⋆) variant  defines new keys:
\ltxkeys@definekeys⋆[KV]{fam}[mp@]{%
  % Command key with callback:
  keya={keepbraced}/\def\x##1{##1*#1*##1};
  % Boolean key:
  keyb=true/\def\y##1{##1yyy#1};
  % Command key with no callback:
  keyc=xxx;
  % Choice-like command key:
  keyd=center/\CheckUserInput{#1}{left,right,center}
    \ifinputvalid
      \edef\myval{\expandcsonce\userinput}
      \edef\numberinlist{\number\order}
      \edef\mychoices{\expandcsonce\nominations}
    \else
      \@latex@error{Input '#1' not valid}\@ehd
    \fi;
  % Boolean key with no callback:
  keye=false;
}
```

In this example, \userinput corresponds to #1; \order is the numerical order of the user input in \nominations; the list of valid values suggested at key definition time ({left | right | center} in this example). The boolean inputvalid is associated with the command \CheckUserInput and is available to the user. It is set true when the user input is valid, and false otherwise. The command \CheckUserInput expects two arguments: the user input and the list of nominations. It doesn't expect two branches (see subsection 16.2).

3.8 Defining all types of key with one command

New macro: \ltxkeys@declarekeys

```
\ltxkeys@declarekeys[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{%
  ⟨keytype⟩/⟨keyname⟩/⟨dft⟩/⟨cbk⟩;
  another set of key attributes;
  etc.
}
\ltxkeys@declarekeys⋆[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{%
  ⟨keytype⟩/⟨keyname⟩/⟨dft⟩/⟨cbk⟩;
  another set of key attributes;
  etc.
}
```

Here, the default value ⟨dft⟩ and the callback ⟨cbk⟩ can be absent in all cases. ⟨keytype⟩ may be any one of {ord, cmd, sty, sty*, bool, choice}. The star (⋆) in 'sty⋆' has the same meaning as

in \ltxkeys@stylekey above, namely, undefined dependants will be defined on the fly when the parent key is set. The optional quantity ⟨mp⟩ is the macro prefix, as in, e. g., subsection 3.3.

Choice keys must have their names associated with their admissible ⟨alt⟩ values in the format ⟨keyname⟩.⟨{alt}⟩ (see example below).

The starred (⋆) variant of \ltxkeys@declarekeys can be used to define new keys (in the sense of \newcommand).

Note 3.5 Keys defined by \ltxkeys@declarekeys are automatically set instantly with their default values, to provide default functions for immediate use. Boolean keys are always initialized in this sense with 'false', so that they aren't turned 'true' prematurely.

Examples: \ltxkeys@declarekeys

```
\ltxkeys@declarekeys*[KV]{fam}[mp@]{%
  % Ordinary key with callback:
  ord/keya/.1\paperwidth/\leftmargin=#1\relax;
  % Command key with callback. '.code=' is allowed before callback:
  cmd/keyb/10mm/.code=\rightmargin=#1\def\x##1{##1*#1*##1};
  % Boolean key without callback:
  bool/keyc/true;
  % Boolean key with callback:
  bool/keyd/true/\ifmp@keyd\@tempswatrue\else\@tempswafalse\fi;
  % Style key with callback but no dependants:
  sty/keye/aaa/.code=\def\y##1{##1yyy#1};
  % Style key with callback and dependants 'keyg' and 'keyh':
  sty*/keyf/blue/\def\y##1{##1#1}/
    cmd>keyg>\parentval>\def\z####1{####1+##1+####1},
    ord>keyh>\KV@fam@keyf@value;
  % Choice key with simple nominations and callback. The function
  % \order is generated internally:
  choice/keyi.{left,right,center}/center/
    \edef\shoot{\ifcase\order 0\or 1\or 2\fi};
  % Choice key with complex nominations:
  choice/keyj.{
      center/.code=\def\mp@textalign{center},
      left/.code=\def\mp@textalign{flushleft},
      % '.code=' can be omitted:
      right/\def\mp@textalign{flushright},
      justified/\let\mp@textalign\relax
    }
    /center/\def\yy##1{##1yy#1};
  ord/keyk/\letcstocsn\func{as-defined-by-user}
}
```

Notice the >...> used for the attributes of the dependant keys 'keyg' and 'keyh' of style key 'keyf'. Dependent keys come as the last attributes of a style key, and they (dependant keys) are separated by comma ','. The default value of the dependant key 'keyg' will in this example be whatever is submitted for 'keyf'. As indicated in subsection 3.4, the function \KV@fam@keyf@value has a longer shelf life than \parentval. Notice also the syntax ⟨keyi⟩.⟨{left,right,center}⟩ for the choice keys 'keyi' and 'keyj'. It says that the alternate admissible values for 'keyi' are 'left', 'right', 'center' and 'justified'; similarly for key 'keyj'.

3.8.1 Defining keys of common type with \ltxkeys@declarekeys

If you have to define keys of the same type with the command \ltxkeys@declarekeys, then the following syntax allows you to avoid entering the key types repeatedly:

Macro: \ltxkeys@declarekeys

```
\ltxkeys@declarekeys(⟨keytype⟩)[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{%
  ⟨keyname⟩/⟨dft⟩/⟨cbk⟩;
  another set of key; etc.
}
\ltxkeys@declarekeys*(⟨keytype⟩)[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{%
  ⟨keyname⟩/⟨dft⟩/⟨cbk⟩;
  another set of key; etc.
}
```

Examples: \ltxkeys@declarekeys

```
\ltxkeys@declarekeys(bool)[KV]{fam}[mp@]{%
  keya/true/\def\x##1{##1*##1*##1};
  keyb/true;
  keyc/true/\def\y##1{##1yyy#1}
}
\ltxkeys@declarekeys*(sty*)[KV]{fam}[mp@]{%
  keyd/xxx/\def\y##1{##1yyy#1};
  % keyf is a dependant of keye:
  keye/blue/\def\y##1{##1#1}/cmd>keyf>\parentval>\def\z####1{####1+##1+####1}
}
```

3.9 Need-value keys

Sometimes you may want to create keys for which the user must always supply his/her own values, even if the keys originally have default values. The default values of keys may not always be suitable. Take, for example, the height and width of a graphics image. For functions that are meant to handle generic images, it would certainly be inappropriate to relieve the user of the need to call picture height and width without corresponding values.

To make a key a need-value key, simply attach the pointer \needvalue to the key at definition time. This pointer can be used only when defining keys, and not when setting keys.

Need-value keys

```
\ltxkeys@cmdkey[KV]{fam}[mp@]{\needvalue{keya}}[blue]{%
  \def\x##1{##1x#1x##1}%
}
\ltxkeys@setkeys[KV]{fam}{keya}
%  -> Error: the author of 'keya' designed it to require a user value.
```

See more about key pointers in subsection 4.4.

3.10 Cross-family keys

There are times when it is required to use the same, or nearly the same, set of keys for different functions and purposes, and thus for different key families and prefixes. We call such keys 'cross-family keys' or 'xfamily keys'. Such keys bear the same names across key families and key prefixes.

For example, the xwatermark package defines three functions (\xwmminipage, \xwmboxedminipage and \xwmcolorbox) using nearly the same set of keys. In each of the three families, the keys bear the same or similar names and they have similar callbacks. The management of cross-family keys can be simplified by using the tools of this section. Even if not all the cross-family keys are needed in all the families to which they may belong, there are still advantages in using this type of keys when some of the keys cut across families.

Cross-family keys are automatically initialized after being defined—as we saw in the case of the commands \ltxkeys@definekeys and \ltxkeys@declarekeys.

New macros: \ltxkeys@savexfamilykeys, \ltxkeys@definexfamilykeys

```
\ltxkeys@savexfamilykeys<⟨id⟩>{⟨keylist⟩}
\ltxkeys@savexfamilykeys*<⟨id⟩>⟨keylistcmd⟩

\ltxkeys@savexfamilykeys<⟨id⟩>(⟨keytype⟩){⟨keylist⟩}
\ltxkeys@savexfamilykeys*<⟨id⟩>(⟨keytype⟩)⟨keylistcmd⟩

\ltxkeys@definexfamilykeys<⟨id⟩>[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨na⟩}
\ltxkeys@definexfamilykeys*<⟨id⟩>[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨na⟩}
```

Here, ⟨id⟩ is the mandatory identifier of the key list ⟨keylist⟩, ⟨pref⟩ is the key prefix, ⟨fam⟩ the key family, ⟨mp⟩ is the macro prefix, and ⟨na⟩ is the list of keys belonging to ⟨keylist⟩ that shouldn't be presently defined and initialized. The ⟨na⟩ can be empty, but it must always be there as a mandatory argument. *So, where you put the key list in the commands* \ltxkeys@definekeys *and* \ltxkeys@declarekeys *is where you now have to locate* ⟨*na*⟩. For any use of the command \ltxkeys@definexfamilykeys we expect the ⟨na⟩ to be far less than the remaining keys. The starred (⋆) variant of \ltxkeys@savexfamilykeys will expand ⟨keylistcmd⟩ once before saving the xfamily keys. The starred (⋆) variant of \ltxkeys@definexfamilykeys will define only definable keys, in the sense of \newcommand.

⟨keylist⟩ and ⟨keylistcmd⟩ have the same syntax as the last arguments of \ltxkeys@definekeys and \ltxkeys@declarekeys:

Syntax of keylist

```
⟨keytype⟩/⟨keyname⟩/⟨dft⟩/⟨cbk⟩;
another set of key attributes;
etc.
```

Here too ⟨keytype⟩ must be a member of the set {ord, cmd, sty, sty*, bool, choice}, ⟨keyname⟩ is obviously the name of the key, ⟨dft⟩ is the default value of the key, and ⟨cbk⟩ is the callback of the key. If the key is a style key, you can add the attributes of the dependants after ⟨cbk⟩ (see the syntaxes of the commands \ltxkeys@definekeys and \ltxkeys@declarekeys).

The mandatory identifier ⟨id⟩ for each list must be unique, not withstanding the fact that the identifiers have their separate namespace.

If the xfamily keys are all of the same type (i. e., only one of the types {ord, cmd, sty, sty*, bool, choice}), you can specify ⟨keytype⟩ as an optional argument in parenthesis to the command \ltxkeys@savexfamilykeys. The parenthesis can't appear with an empty content.

Examples: xfamily keys

```
\ltxkeys@savexfamilykeys<x1>{%
  ord/keya/\paperwidth/\mylength=#1;
  cmd/keyb/black/\def\y##1{##1};
```

```
  choice/keyc.{left,right,center}/center/\def\z##1{##1};
  bool/keyd/true
}

% Now define the keys previously stored with the id no. x1.
% For now don't define keys keyb and keyc:
\ltxkeys@definexfamilykeys<x1>[KV]{fam}[mp@]{keyb,keyc}

% Once defined the keys can be executed separately:
\ltxkeys@setkeys[KV]{fam}{keya=.5\hsize,keyd=false}
\show\ifmp@keyd

% Now define the keys previously stored with the id no. x1 for
% another family. This time we don't want to define key keyb:
\ltxkeys@definexfamilykeys<x1>[KVA]{fama}[mpa@]{keyb}

% You can save and define xfamily keys of only one key type,
% command keys in the following example:
\ltxkeys@savexfamilykeys<x1>(cmd){%
  keya/\paperwidth;
  keyb/blue/\def\x##1{#1x##1};
}
% Define the saved keys and ignore none of them:
\ltxkeys@definexfamilykeys*<x1>[KV]{fam}[mp@]{}
\ltxkeys@setkeys[KV]{fam}{keya=.5\hsize,keyb=red}
```

Examples: xfamily keys

```
% 'keya' and 'keyd' are starred style keys but 'keyd' has no dependants:
\ltxkeys@savexfamilykeys<a1>(sty*){%
  keya/center/.code=\def\xx##1{##1xx#1}/
    ord>\needvalue{keyb}>\parentval>\edef\yy##1{##1yy\unexpanded{#1}},
    % The braces around 'center' (the default value of 'keyc')
    % will be preserved in parsing:
    cmd>keyc>{center};
  % The braces around the callback of 'keyd' will be preserved:
  keyd/red/.code={\def\x{\color{#1}print aaa}};
}
% Ignore 'keyd' in defining keys saved in 'a1':
\ltxkeys@definexfamilykeys*<a1>[KV]{fam}[mp@]{keyd}
% On setting 'keya', 'keyb' and 'keyc' will be defined and initialized:
\ltxkeys@setkeys[KV]{fam}{keya=left}
```

Here is a real-life example that mimics some of the macros of the xwatermark package:

Examples: xfamily keys

```
\ltxkeys@savexfamilykeys<a1>{%
  cmd/width/\textwidth;
  cmd/textcolor/black;
  cmd/framecolor/black;
  cmd/framesep/3\p@;
  cmd/framerule/0.4\p@;
  choice/textalign.{%
```

```
      center/.code=\def\mp@textalign{center},
      left/.code=\def\mp@textalign{flushleft},
      right/.code=\def\mp@textalign{flushright}
    }/center;
  bool/framebox/true;
  ord/junkkey/throwaway;
}
% Ignore keys ‘framebox’ and ‘junkkey’ when defining family ‘ltxframebox’:
\ltxkeys@definexfamilykeys*<a1>[KV]{ltxframebox}[mp@]{framebox,junkkey}
% Ignore key ‘junkkey’ when defining family ‘ltxminipage’:
\ltxkeys@definexfamilykeys<a1>[KV]{ltxminipage}[mp@]{junkkey}
% No key is ignored when defining ‘junkfamily’:
\ltxkeys@definexfamilykeys<a1>[KVX]{junkfamily}[mp@]{}

\newcommand*\ltxframebox[2][]{%
  \ltxkeys@setkeys[KV]{ltxframebox}{#1}%
  \begingroup
  \fboxsep\mp@framesep\fboxrule\mp@framerule
  \cptdimdef\mp@boxwidth{\mp@width-2\fboxsep-2\fboxrule}%
  \color{\mp@framecolor}%
  \noindent
  \fbox{%
    \removelastskip
    \parbox{\mp@boxwidth}{%
      \begin\mp@textalign
      \textcolor{\mp@textcolor}{#2}%
      \end\mp@textalign
    }%
  }%
  \endgroup
}
\newcommand*\ltxminipage[2][]{%
  \ltxkeys@setkeys[KV]{ltxminipage}{#1}%
  \begingroup
  \fboxsep\mp@framesep
  \fboxrule\ifmp@framebox\mp@framerule\else\z@\fi
  \cptdimdef\mp@boxwidth{\mp@width-2\fboxsep-2\fboxrule}%
  \noindent\begin{lrbox}\@tempboxa
  \begin{minipage}[c][\height][s]\mp@boxwidth
  \@killglue
  \begin\mp@textalign
  \textcolor{\mp@textcolor}{#2}%
  \end\mp@textalign
  \end{minipage}%
  \end{lrbox}%
  \@killglue
  \color{\mp@framecolor}%
  \ifmp@framebox\fbox{\fi\usebox\@tempboxa\ifmp@framebox}\fi
  \endgroup
}

\begin{document}
\ltxframebox[
```

```
357   framecolor=blue,textcolor=purple,textalign=left
358 ]{%
359   Test text\endgraf ...\endgraf test text
360 }
361 \medskip
362 \ltxminipage[
363   framecolor=blue,textcolor=purple,framebox=true,textalign=right
364 ]{%
365   Test text\endgraf ...\endgraf test text
366 }
367 \end{document}
```

4 Setting keys

In the ltxkeys package there are many functions for setting keys. Keys can be set by the following utilities.

4.1 Setting defined keys

New macros: \ltxkeys@setkeys

```
368 \ltxkeys@setkeys[⟨pref⟩]{⟨fam⟩}[⟨na⟩]{⟨keyval⟩}
369 \ltxkeys@setkeys*[⟨pref⟩]{⟨fam⟩}[⟨na⟩]{⟨keyval⟩}
370 \ltxkeys@setkeys+[⟨prefs⟩]{⟨fams⟩}[⟨na⟩]{⟨keyval⟩}
371 \ltxkeys@setkeys*+[⟨prefs⟩]{⟨fams⟩}[⟨na⟩]{⟨keyval⟩}
```

Here, ⟨prefs⟩, ⟨fams⟩ and ⟨keyval⟩ are comma-separated list of key prefixes, families and ⟨key⟩=⟨value⟩ pairs, respectively. Keys listed in the comma-separated list ⟨na⟩ are ignored. The starred (*) variant will save all undefined keys with prefix ⟨pref⟩ and in family ⟨fam⟩ in the macro \⟨pref⟩@⟨fam⟩@⟨rmkeys⟩, to be set later, perhaps with \ltxkeys@setrmkeys. The plus (+) variant will search in all the prefixes in ⟨prefs⟩ and all families in ⟨fams⟩ for a key before logging the key in \⟨pref⟩@⟨fam⟩@⟨rmkeys⟩ (if the *+ variant is used) or reporting it as undefined.

To avoid infinite re-entrance of \ltxkeys@setkeys and the consequent bombing out of the command, the package option keydepthlimit is introduced. Its default value is 4, meaning that \ltxkeys@setkeys can't ordinarily be nested beyond level 4. If you must nest \ltxkeys@setkeys beyond this level, an unlikely need, you can raise the keydepthlimit as a package option via \usepackage or, if catoptions package is loaded before \documentclass, via \documentclass. For example,

Setting keydepthlimit

```
372 \usepackage[keydepthlimit=6]{ltxkeys}
```

The more appropriate name keystacklimit is an alias for keydepthlimit.

4.2 Setting 'remaining' keys

The command \ltxkeys@setrmkeys, which has both star (*) and plus (+) variants, is the counterpart of \setrmkeys of the xkeyval package:

New macro: \ltxkeys@setrmkeys

```
\ltxkeys@setrmkeys[⟨pref⟩]{⟨fam⟩}[⟨na⟩]
\ltxkeys@setrmkeys*[⟨pref⟩]{⟨fam⟩}[⟨na⟩]
\ltxkeys@setrmkeys+[⟨prefs⟩]{⟨fams⟩}[⟨na⟩]
\ltxkeys@setrmkeys*+[⟨prefs⟩]{⟨fams⟩}[⟨na⟩]
```

The command \ltxkeys@setrmkeys sets in the given prefixes and families the 'remaining keys' saved when calling the starred (⋆) variant of \ltxkeys@setkeys or \ltxkeys@setrmkeys. ⟨na⟩ is again the list of keys that should be ignored, i. e., not executed and not saved. The unstarred variant of \ltxkeys@setrmkeys will report an error if a key is undefined. The starred (⋆) variant of the macro \ltxkeys@setrmkeys, like the starred (⋆) variant of \ltxkeys@setkeys, ignores keys that it cannot find and saves them on the list saved for a future call to \ltxkeys@setrmkeys. Keys listed in ⟨na⟩ will be ignored fully and will not be appended to the saved list of remaining keys.

4.3 Setting aliased keys

Aliased keys differ from style keys of subsection 3.4. Two keys may be aliased to each other, such that when one is set, the alias is automatically set with the same or a different value. The concept is similar to, but not identical with, that of style keys. The two aliases must all be in the same family and have the same key and macro prefixes.

New macro: \ltxkeys@setaliaskey

```
\ltxkeys@setaliaskey{⟨key⟩}[⟨value⟩]
```

Here, ⟨value⟩ is optional; if it is not given, ⟨key⟩ will be set with the current value of its alias. The command \setaliaskey is a shortened variant of \ltxkeys@setaliaskey.

Examples: \ltxkeys@setaliaskey

```
\ltxkeys@definekeys*[KV]{fam}[mp@]{%
  printsign=true;
  printmark=true/\ltxkeys@setaliaskey{printsign}[false];
  keya=$+++$;
  keyb=star/\ltxkeys@setaliaskey{keya}[$***$]
}
\ltxkeys@definekeys*[KV]{fam}[mp@]{%
  keya=sun/\CheckUserInput{#1}{star,sun,moon}
    \ifinputvalid
      \edef\givenval{\userinput}
      \edef\found{\ifcase\order star@\or sun@\or moon@\fi}
    \else
      \@latex@error{Input '#1' not valid}\@ehd
    \fi;
  keyb=star/\ltxkeys@setaliaskey{keya};
}
```

The boolean \ifinputvalid associated with the command \CheckUserInput is described in macro line 179 (see also subsection 16.2).

The example involving 'printsign' and 'printmark' is similar, but not equivalent, to the notion of biboolean keys. Biboolean keys have equal symmetry (i. e., they can call each other with equal propensity) and they won't bomb out in an infinite reentrance. This is not the case with aliased keys: only slave/alias can set or call master/main key. If they both call each other, the user will

be alerted to the fact that there is an infinite reentrance of keys. The notion of 'slave' and 'master' used in the ltxkeys package may be counterintuitive but in reality it is quite logical.

Schemes like the following are disallowed, to avoid back-linking of \ltxkeys@setaliaskey. The package will flag an error if something like the following occurs:

Examples: Illegal nested \ltxkeys@setaliaskey

```
\ltxkeys@ordkey[KV]{fam}{keya}[true]{\setaliaskey{keyb}}
\ltxkeys@ordkey[KV]{fam}{keyb}[true]{\setaliaskey{keya}}
\ltxkeys@setkeys[KV]{fam}{keya}
```

4.4 Using key pointers

The \savevalue and \usevalue pointers of the xkeyval package are still available at key setting time, but with increased robustness and optimization. Curly braces in values are preserved throughout, and instead of saving the value of each key tagged with \savevalue in a separate macro, we save all such keys and their values in only one macro (for each combination of ⟨pref⟩ and ⟨fam⟩) and use a fast search technique to find the values when they are needed later (by any key tagged with \usevalue).

The pointer \needvalue is a new type. It can be used by any key author to prompt the user of the key to always supply a value for the key. The pointers \savevalue, \usevalue and \needvalue can all be used when defining keys; the pointer \usevalue will, however, be ignored when defining keys, until when setting keys. The pointers \savevalue and \usevalue can both be used when setting keys, but not the pointer \needvalue.

Here is an interesting example and proof of concept of pointers:

Key pointers

```
\ltxkeys@stylekeys*[KV]{fam}{%
  \needvalue{keya},\savevalue\needvalue{keyb},\needvalue\savevalue{keyc}
}[{left}](%
  % '#1' here refers to the value of the dependant key at the
  % time it is being set.
  ord/\savevalue{keyb}/\parentval/\edef\y##1{##1xx\unexpanded{#1}};
  cmd/keyc/{center}
){%
  % '#1' here refers to the value of the parent key at the time
  % it is being set.
  \def\x##1{##1xx#1}
}

\ltxkeys@setkeys[KV]{fam}{%
  \savevalue{keya}={\def\y##1{##1}},
  \savevalue{keyb}=\usevalue{keya},
  keyc=\usevalue{keyb}
}
```

If you have to save the values of many keys, then the above scheme of placing \savevalue on keys at key setting time can be avoided by using the following commands:

New macros: \ltxkeys@savevaluekeys, \ltxkeys@addsavevaluekeys, etc

```
\ltxkeys@savevaluekeys[⟨pref⟩]{⟨fam⟩}{⟨list⟩}
\ltxkeys@addsavevaluekeys[⟨pref⟩]{⟨fam⟩}{⟨list⟩}
```

```
\ltxkeys@removesavevaluekeys[⟨pref⟩]{⟨fam⟩}{⟨list⟩}
\ltxkeys@undefsavevaluekeys[⟨pref⟩]{⟨fam⟩}
\ltxkeys@undefsavevaluekeys![⟨pref⟩]{⟨fam⟩}
\ltxkeys@emptifysavevaluekeys[⟨pref⟩]{⟨fam⟩}
\ltxkeys@emptifysavevaluekeys![⟨pref⟩]{⟨fam⟩}
```

The command \ltxkeys@savevaluekeys will create, for the given key family and prefix, a list of keys whose values should be saved at key-setting time, if those keys don't already exist in the list. The command \ltxkeys@addsavevaluekeys will add to the list those keys that don't already exist in the list; \ltxkeys@removesavevaluekeys remove those save-keys that it can find in the list; while the command \ltxkeys@undefsavevaluekeys will undefine the entire list of save-keys of the given key family and prefix. The command \ltxkeys@emptifysavevaluekeys will simplify emptify the content of the save-key list. The ! variant of the commands

Macros

```
\ltxkeys@undefsavevaluekeys
\ltxkeys@emptifysavevaluekeys
```

will undefine or emptify the existing save-key list globally.

Examples: \ltxkeys@savevaluekeys

```
\ltxkeys@definekeys[KV]{fam}[mp@]{%
  ord/keya/2cm/\def\x##1{#1xx##1};
  cmd/keyb/John;
  bool/keyc/true/\ifmp@keyc\def\y##1{##1yy#1}\fi;
  choice/keyd.{left,right,center}/
    \ifcase\order\def\shoot{0}\or\def\shoot{1}\or\def\shoot{2}\fi
}

\ltxkeys@savevaluekeys[KV]{fam}{keya,keyb,keyc}
\ltxkeys@addsavevaluekeys[KV]{fam}{keyd}
\ltxkeys@removesavevaluekeys[KV]{fam}{keya,keyb}
\ltxkeys@undefsavevaluekeys[KV]{fam}

\ltxkeys@setkeys[KV]{fam}{keya=\usevalue{keyc},keyb=\usevalue{keya}}
```

4.5 Accessing the saved value of a key

As mentioned earlier, the pointers \savevalue and \usevalue are available for saving and using the values of keys within the command \ltxkeys@setkeys. But suppose you have used \savevalue within \ltxkeys@setkeys to set the value of a key, how do you access that value outside of \ltxkeys@setkeys? You can do this by using the following \ltxkeys@storevalue command:

New macro: \ltxkeys@storevalue

```
\ltxkeys@storevalue[⟨pref⟩]{⟨fam⟩}{⟨key⟩}⟨cs⟩
\ltxkeys@storevalue+[⟨pref⟩]{⟨fam⟩}{⟨key⟩}⟨cs⟩⟨fallback⟩
```

Here, ⟨cs⟩ is the macro (defined or undefined) that will receive the saved value of ⟨key⟩. The plain variant of this command will raise an error message if the value of the key wasn't previously saved, while the plus (+) variant will resort to the user-supplied function ⟨fallback⟩. Only saved key values can be recovered by this command.

Examples: \ltxkeys@storevalue

```
\ltxkeys@cmdkey[KV]{fam}{\needvalue{keya}}[{left}]{%
  \def\x##1{##1xx#1}
}
\ltxkeys@setkeys[KV]{fam}{\savevalue{keya}={\def\y##1{##1}}}
\ltxkeys@storevalue[KV]{fam}{keya}\tempa
\ltxkeys@storevalue+[KV]{fam}{keya}\tempb{%
  \@latex@error{No value saved for key 'keya'}\@ehc
}
```

4.6 Pre-setting and post-setting keys

New macros: \ltxkeys@presetkeys, \ltxkeys@postsetkeys, etc.

```
\ltxkeys@presetkeys[⟨pref⟩]{⟨fam⟩}{⟨keys⟩}
\ltxkeys@presetkeys![⟨pref⟩]{⟨fam⟩}{⟨keys⟩}
\ltxkeys@addpresetkeys[⟨pref⟩]{⟨fam⟩}{⟨keys⟩}
\ltxkeys@addpresetkeys![⟨pref⟩]{⟨fam⟩}{⟨keys⟩}
\ltxkeys@removepresetkeys[⟨pref⟩]{⟨fam⟩}{⟨keys⟩}
\ltxkeys@removepresetkeys![⟨pref⟩]{⟨fam⟩}{⟨keys⟩}
\ltxkeys@undefpresetkeys[⟨pref⟩]{⟨fam⟩}
\ltxkeys@undefpresetkeys![⟨pref⟩]{⟨fam⟩}

\ltxkeys@postsetkeys[⟨pref⟩]{⟨fam⟩}{⟨keys⟩}
\ltxkeys@postsetkeys![⟨pref⟩]{⟨fam⟩}{⟨keys⟩}
\ltxkeys@addpostsetkeys[⟨pref⟩]{⟨fam⟩}{⟨keys⟩}
\ltxkeys@addpostsetkeys![⟨pref⟩]{⟨fam⟩}{⟨keys⟩}
\ltxkeys@removepostsetkeys[⟨pref⟩]{⟨fam⟩}{⟨keys⟩}
\ltxkeys@removepostsetkeys![⟨pref⟩]{⟨fam⟩}{⟨keys⟩}
\ltxkeys@undefpostsetkeys[⟨pref⟩]{⟨fam⟩}
\ltxkeys@undefpostsetkeys![⟨pref⟩]{⟨fam⟩}
```

The optional ! here, as in many instances in the ltxkeys package, means that the assignments would be done (and the lists built) globally rather than locally. 'Presetting keys' means 'these keys should be set before setting other keys in every run of the command \ltxkeys@setkeys for the given key prefix and family'. \ltxkeys@addpresetkeys is an alias for \ltxkeys@presetkeys, and this helps explain that \ltxkeys@presetkeys is indeed a list merger. Neither the command \ltxkeys@presetkeys nor \ltxkeys@postsetkeys set keys itself, contrary to what the names might suggest. 'Post-setting keys' means 'these keys are to be set after setting other keys in every run of \ltxkeys@setkeys for the given key prefix and family'. \ltxkeys@addpostsetkeys is an alias for \ltxkeys@postsetkeys. The commands

Macros

```
\ltxkeys@removepresetkeys![⟨pref⟩]{⟨fam⟩}{⟨keys⟩}
\ltxkeys@removepostsetkeys![⟨pref⟩]{⟨fam⟩}{⟨keys⟩}
```

remove ⟨keys⟩ from preset and post-set lists, respectively. The commands

Macros

```
\ltxkeys@undefpresetkeys![⟨pref⟩]{⟨fam⟩}
\ltxkeys@undefpostsetkeys![⟨pref⟩]{⟨fam⟩}
```

respectively, undefine all preset and post-set keys in the given family.

Logically, you can't enter the same key twice in either preset or post-set list in the same family and prefix.

Examples: \ltxkeys@presetkeys, \ltxkeys@postsetkeys, etc.

```
\ltxkeys@definekeys*[KV1]{fam1}[mp@]{%
  keya/left/\def\x##1{#1x##1};
  \needvalue{keyb}/right;
  keyc/center;
  keyd
}
\ltxkeys@presetkeys![KV1]{fam1}{keya=\flushleft,keyb=\flushright}
\ltxkeys@postsetkeys![KV1]{fam1}{keyd=\flushleft}
...
% Eventually, only 'keya' will be preset:
\ltxkeys@removepresetkeys![KV1]{fam1}{keyb=\flushright}
...
% Because of the *  and +  signs on \ltxkeys@setkeys, all unknown
% keys (those with prefix 'KV2' and in family 'fam2') will be saved in
% the list of remaining keys, and can be set later with \ltxkeys@setrmkeys:
\ltxkeys@setkeys*+[KV1,KV2]{fam1,fam2}[keyd]{keya=xxx,keyb=yyy,keyc}
```

4.7 Initializing keys

New macro: \ltxkeys@initializekeys

```
\ltxkeys@initializekeys[⟨prefs⟩]{⟨fams⟩}[⟨na⟩]
```

This presets all the keys previously defined in families ⟨fams⟩ with their default values; it ignores keys listed in ⟨na⟩. If ⟨na⟩ is a list of ⟨key⟩=⟨value⟩ pairs, the key names are extracted from the list before the family keys are initialized. Any ⟨key⟩=⟨value⟩ pairs in ⟨na⟩ are not set at all. All keys defined by \ltxkeys@definekeys and \ltxkeys@declarekeys are automatically instantly initialized, except slave/alias and dependant keys. Alias and dependant keys aren't initialized in this case in order to avoid cyclic re-entrance of \ltxkeys@setkeys.

The command \ltxkeys@initializekeys can be used in place of \ltxkeys@executeoptions, since \ltxkeys@executeoptions (similar to LaTeX kernel's \ExecuteOptions) fulfils the sole purpose of setting up default values of options. Keys defined via \ltxkeys@definekeys and \ltxkeys@declarekeys don't have to be initialized, since they're automatically initialized at definition time.

Note 4.1 Keys that have been processed by \ltxkeys@processoptions (i. e., keys submitted by the user as package or class options via \documentclass or \usepackage can't be initialized or launched (see subsection 4.8 below for the meaning of 'launched keys'). This is to avoid unwittingly setting keys to their default values after the user has submitted them as package or class options. This means that 'option keys' (see section 7) can't be initialized or launched.

4.8 Launching keys

New macro: \ltxkeys@launchkeys

```
\ltxkeys@launchkeys[⟨prefs⟩]{⟨fams⟩}{⟨curr⟩}
\ltxkeys@launchkeys*[⟨prefs⟩]{⟨fams⟩}{⟨curr⟩}
```

```
\ltxkeys@launchkeys+[⟨prefs⟩]{⟨fams⟩}{⟨curr⟩}
\ltxkeys@launchkeys*+[⟨prefs⟩]{⟨fams⟩}{⟨curr⟩}
```

This presets all keys defined in families ⟨fams⟩ with their default values; it ignores keys listed in ⟨curr⟩. ⟨curr⟩ may be the list of ⟨key⟩=⟨value⟩ pairs that the user wants to use as current values of keys. Their keys are to be ignored when setting up defaults, i. e., when initializing the family keys. One major difference between \ltxkeys@launchkeys and \ltxkeys@initializekeys is that in \ltxkeys@launchkeys the ⟨key⟩=⟨value⟩ pairs in ⟨curr⟩ are immediately set after the absent family keys (i. e., those without current values) are reinitialized. Keys appearing in ⟨curr⟩ in the command \ltxkeys@launchkeys will be the ⟨na⟩ (ignored) keys for the command \ltxkeys@initializekeys.

Keys across multiple prefixes ⟨prefs⟩ and families ⟨fams⟩ can be launched at the same time, but the user has to know what is he doing: the keys might not have been defined across the given families, or some keys might have been disabled in some, and not all, families. The * and + variants of \ltxkeys@launchkeys have the same meaning as in \ltxkeys@setkeys (section 4). The starred (*) variant will save all undefined keys with prefix ⟨pref⟩ and in family ⟨fam⟩ in the macro \⟨pref⟩@⟨fam⟩@⟨rmkeys⟩, to be set later, perhaps with the command \ltxkeys@setrmkeys. The plus (+) variant will search in all the prefixes in ⟨prefs⟩ and all families in ⟨fams⟩ for a key before logging the key in \⟨pref⟩@⟨fam⟩@⟨rmkeys⟩ (if the *+ variant is the one used) or reporting it as undefined.

4.8.1 Non-initialize and non-launch keys

Listing all the keys that shouldn't be reinitialized by \ltxkeys@initializekeys in the ⟨na⟩ list every time \ltxkeys@initializekeys is called can sometimes be inconvenient, especially when dealing with a large number of keys. Perhaps even more important is the fact that sometimes you don't want some of the keys in a family to be reinitialized even though they are absent keys (i. e., they aren't listed as current keys, meaning that they aren't in the current ⟨key⟩=⟨value⟩ list submitted to \ltxkeys@launchkeys). This might be the case with package and class options. The command \ltxkeys@nonlaunchkeys provides a convenient means for listing the non-reinitializing keys once and for all. If there are keys in a family that shouldn't be reinitialized/launched with other keys in the same family during any call to \ltxkeys@launchkeys or \ltxkeys@initializekeys, they can be listed in the \ltxkeys@nonlaunchkeys command:

New macro: \ltxkeys@nonlaunchkeys

```
\ltxkeys@nonlaunchkeys[⟨prefs⟩]{⟨fams⟩}{⟨keys⟩}
```

Keys across multiple prefixes and families can be submitted to the \ltxkeys@nonlaunchkeys command: undefined keys are simply ignored by \ltxkeys@nonlaunchkeys.

Note 4.2 The command \ltxkeys@nonlaunchkeys doesn't mean that the keys in ⟨keys⟩ can no longer be set via the command \ltxkeys@setkeys; it simply implies that keys appearing in \ltxkeys@nonlaunchkeys will not be reinitialized to their default values when members of their class are being launched or reinitialized. The command \ltxkeys@noninitializekeys is an alias for \ltxkeys@nonlaunchkeys.

4.9 Handling unknown keys and options

You can use the macro \ltxkeys@unknownkeyhandler to declare to the ltxkeys package the course of action to take if, while setting keys, it discovers that a key is undefined or unknown. The

command \ltxkeys@unknownoptionhandler applies to unknown options (see section 11)[*7]. The syntax of these commands is

New macros: \ltxkeys@unknownkeyhandler, \ltxkeys@unknownoptionhandler

```
\ltxkeys@unknownkeyhandler[⟨prefs⟩]{⟨fams⟩}{⟨cbk⟩}
\ltxkeys@unknownoptionhandler[⟨pref⟩]<⟨fam⟩>{⟨cbk⟩}
```

The callback ⟨cbk⟩ signifies the action to take when an unknown key or option is encountered. The default ⟨cbk⟩ is to log the keys and, in each run, warn the user of the presence of unknown keys. The same ⟨cbk⟩ can be used across key prefixes ⟨prefs⟩ and families ⟨fams⟩. You can use #1 (or \CurrentPref) in this macro to represent the current key prefix, #2 (or \CurrentFam) for the current family, #3 (or \CurrentKey) for the current key name, and #4 (or \CurrentVal) for the value of the current key. If \CurrentVal contains undefined macros or active characters, then attempting to print it may cause problems. Therefore, when making entries in the transcript file, it will sometimes be preferable to use \InnocentVal instead of \CurrentVal. However, \InnocentVal will give only the first eight characters of a key's value.

The following example provides an unknown key handler for two key prefixes (KVA and KVB) and two key families (fam1 and fam2):

Examples: \ltxkeys@unknownkeyhandler

```
\ltxkeys@unknownkeyhandler[KVA,KVB]{fam1,fam2}{%
  \@expandtwoargs\in@{,#3,}{,\myspecialkeys,}%
  \ifin@
    % The reader may want to investigate what the parameter texts
    % ##1 and ####1 below stand for (see note 4.3 below):
    \ltxkeys@ordkey[#1]{#2}{#3}[#4]{\def\x####1{####1xx##1}}%
  \else
    \ltxmsg@warn{Unknown key ‘#3’ with value ‘#4’ in family ‘#2’ ignored}\@ehd
    % \ltxmsg@warn{Unknown key ‘\CurrentKey’ with value
    %   ‘\InnocentVal’ in family ‘\CurrentFam’ ignored}\@ehd
  \fi
}
```

The macro \myspecialkeys in the above example doesn't actually exist; it is only meant for illustration here. But 'handled keys' may be introduced by the user to serve this purpose. This will be the set of keys for which special actions may apply at key setting time (see section 8).

Note 4.3 To see what the parameter texts ##1 and ####1 above stand for, run the following code on your own and note the outcome of \show\KV@fam@keyd. The characters ##1 will turn out to be the parameter text which can be used to access the current values of keys keyd and keye after they have been defined on the fly. And ####1 will be the parameter text of the arbitrary function \x. If you do \show\KV@fam@keyd, you'll notice that the parameter texts have been reduced by one level of nesting.

Examples: \ltxkeys@unknownkeyhandler

```
\def\myspecialkeys{keyc,keyd,keye}
\ltxkeys@unknownkeyhandler[KV]{fam}{%
```

[*7] Options are also keys, but (from the user's viewpoint) there might be a need to treat options separately when dealing with unknown keys.

```
  \@expandtwoargs\in@{,#3,}{,\myspecialkeys,}%
  \ifin@
    \ltxkeys@ordkey[#1]{#2}{#3}[#4]{\def\x####1{####1xx##1}}%
  \else
    \ltxmsg@warn{Unknown key '#3' with value '\InnocentVal'
      in family '#2' ignored}\@ehd
  \fi
}
\ltxkeys@setkeys[KV]{fam}{keyd=aaa,keye=bbb}
\show\KV@fam@keyd
```

5 Checking if a key is defined

New macros: \ltxkeys@ifkeydefTF, \ltxkeys@ifkeydefFT

```
\ltxkeys@ifkeydefTF[⟨prefs⟩]{⟨fams⟩}{⟨key⟩}{⟨true⟩}{⟨false⟩}
\ltxkeys@ifkeydefFT[⟨prefs⟩]{⟨fams⟩}{⟨key⟩}{⟨false⟩}{⟨true⟩}
```

These check if ⟨key⟩ is defined with a prefix in ⟨prefs⟩ and in family in ⟨fams⟩. If the test proves that ⟨key⟩ is defined, ⟨true⟩ text will be executed; otherwise ⟨false⟩ will be executed.

6 Disabling keys

New macro: \ltxkeys@disablekeys

```
\ltxkeys@disablekeys[⟨prefs⟩]{⟨fams⟩}{⟨keys⟩}
\ltxkeys@disablekeys⋆[⟨prefs⟩]{⟨fams⟩}{⟨keys⟩}
```

Here, ⟨keys⟩ is a comma-separated list of keys to be disabled. The macro \ltxkeys@disablekeys causes an error to be issued when a disabled key is invoked. If the package option tracingkeys is true, undefined keys are highlighted by \ltxkeys@disablekeys with a warning message. Because it is possible to mix prefixes and families in \ltxkeys@disablekeys, undefined keys may readily be encountered when disabling keys. To see those undefined keys in the transcript file, enable the package option tracingkeys.

The plain form of \ltxkeys@disablekeys disables the given keys instantly, while the starred (⋆) variant disables the keys at \AtBeginDocument. Authors can use this command to bar users of their keys from calling those keys after a certain point.

7 Option and non-option keys

Sometimes you want to create keys that can only appear in \documentclass, \RequirePackage or \usepackage, and at other times you may not want the user to submit a certain set of keys via these commands. The xwatermark package, for example, uses this concept.

New macros: \ltxkeys@optionkeys, \ltxkeys@nonoptionkeys

```
\ltxkeys@optionkeys[⟨pref⟩]{⟨fam⟩}{⟨keys⟩}
\ltxkeys@optionkeys⋆[⟨pref⟩]{⟨fam⟩}{⟨keys⟩}
\ltxkeys@nonoptionkeys[⟨pref⟩]{⟨fam⟩}{⟨keys⟩}
```

Here, ⟨keys⟩ is a comma-separated list of keys to be made option or non-option keys. Keys listed in \ltxkeys@optionkeys can appear only in arguments of \documentclass, \RequirePackage or \usepackage, while keys listed in \ltxkeys@nonoptionkeys can't appear in these macros. The starred (⋆) variant of \ltxkeys@optionkeys is equivalent to \ltxkeys@nonoptionkeys. Only defined keys may appear in \ltxkeys@optionkeys and \ltxkeys@nonoptionkeys.

New macro: \ltxkeys@makeoptionkeys

```
520 \ltxkeys@makeoptionkeys[⟨pref⟩]{⟨fam⟩}
521 \ltxkeys@makeoptionkeys⋆[⟨pref⟩]{⟨fam⟩}
522 \ltxkeys@makenonoptionkeys[⟨pref⟩]{⟨fam⟩}
```

The command \ltxkeys@makeoptionkeys makes all the keys with prefix ⟨pref⟩ and in family ⟨fam⟩ options keys. The command \ltxkeys@makenonoptionkeys does the reverse, i. e., makes the keys non-option keys. The starred (⋆) variant of \ltxkeys@makeoptionkeys is equivalent to \ltxkeys@makenonoptionkeys.

8 Handled keys

As mentioned in subsection 4.9, handled keys are keys defined in a macro that is key-prefix and key-family dependent. They are defined as a list in a macro so that they can be used for future applications, such as deciding if a dependant key of a style key should be defined or redefined on the fly. Handled keys should be defined, or added to, using key prefix, family and key names. You can define or add to handled keys by the following command:

New macro: \ltxkeys@handledkeys

```
523 \ltxkeys@handledkeys[⟨pref⟩]{⟨fam⟩}{⟨list⟩}
```

where ⟨list⟩ is a comma-separated list of key names. This command can be issued more than once for the same key prefix ⟨pref⟩ and family ⟨fam⟩, since the content of ⟨list⟩ is usually merged with the existing list rather than being merely added or overwritten. There is also

New macro: \ltxkeys@addhandledkeys

```
524 \ltxkeys@addhandledkeys[⟨pref⟩]{⟨fam⟩}{⟨list⟩}
```

which is just an alias for \ltxkeys@handledkeys.

Example: \ltxkeys@handledkeys

```
525 \ltxkeys@handledkeys[KVA,KVB]{fam1,fam2}{keya,keyb,keyc}
```

For a given key prefix ⟨pref⟩ and family ⟨fam⟩, you can recall the full list of handled keys (set up earlier by \ltxkeys@handledkeys) by the command

List of handled keys

```
526 \⟨pref⟩@⟨fam⟩@handledkeys
```

You can remove handled keys from a given list of handled keys (in a family) by the following command:

New macro: \ltxkeys@removehandledkeys

```
527 \ltxkeys@removehandledkeys[⟨pref⟩]{⟨fam⟩}{⟨list⟩}
```

Rather than remove individual handled keys from a list, you might prefer or need to simply undefine or 'emptify' the entire list of handled keys in a family. You can do these with the following commands:

New macros: \ltxkeys@undefhandledkeys, \ltxkeys@emptifyhandledkeys

```
528 \ltxkeys@undefhandledkeys[⟨pref⟩]{⟨fam⟩}
529 \ltxkeys@emptifyhandledkeys[⟨pref⟩]{⟨fam⟩}
```

9 Reserving and unreserving key path or bases

By 'key path' we mean the key prefix (default is KV), key family (generally no default), and macro prefix (default is dependent on the type of key). However, when dealing with 'pathkeys' (see section 15) the term excludes the macro prefix. You can reserve key path or bases (i. e., bar future users from using the same path or bases) by the following commands. Once a key family or prefix name has been used, it might be useful barring further use of those names. For example, the ltxkeys package has barred users from defining keys with key family ltxkeys and macro prefix ltxkeys@.

New macros: \ltxkeys@reservekeyprefix, \ltxkeys@reservekeyfamily, etc.

```
530 \ltxkeys@reservekeyprefix{⟨list⟩}
531 \ltxkeys@reservekeyprefix*{⟨list⟩}
532 \ltxkeys@reservekeyfamily{⟨list⟩}
533 \ltxkeys@reservekeyfamily*{⟨list⟩}
534 \ltxkeys@reservemacroprefix{⟨list⟩}
535 \ltxkeys@reservemacroprefix*{⟨list⟩}
```

Here, ⟨list⟩ is a comma-separated list of bases. The starred (⋆) variants of these commands will defer reservation to the end of the current package or class, while the unstarred variants will effect the reservation immediately. As the package or class author you may want to defer the reservation to the end of your package or class.

Users can, at their own risk, override reserved key bases simply by issuing the package boolean option reservenopath. This can be issued in \documentclass, \usepackage or \ltxkeys@options. This might be too drastic for many users and uses. Therefore, the ltxkeys package also provides the following commands that can be used for selectively unreserving currently reserved key bases:

New macros: \ltxkeys@unreservekeyprefix, \ltxkeys@unreservekeyfamily, etc.

```
536 \ltxkeys@unreservekeyprefix{⟨list⟩}
537 \ltxkeys@unreservekeyprefix*{⟨list⟩}
538 \ltxkeys@unreservekeyfamily{⟨list⟩}
539 \ltxkeys@unreservekeyfamily*{⟨list⟩}
540 \ltxkeys@unreservemacroprefix{⟨list⟩}
541 \ltxkeys@unreservemacroprefix*{⟨list⟩}
```

The starred (⋆) variants of these commands will defer action to the end of the current package or class, while the unstarred variants will undo the reservation immediately.

10 Bad key names

Some key names are indeed inadmissible. The ltxkeys considers the following literals, among others, as inadmissible for key names:

Default bad key names

```
.code          ordkey         cmdkey         choicekey      boolkey
handledkeys    presetkeys     postsetkeys    executedkeys   rmkeys
needvalue      savevalue      usevalue       savevaluekeys  needvaluekeys
xkeys          bool           boolean        tog            togg
toggle         switch         true           false          on
off            count          skip
```

For reasons of efficiency, the ltxkeys package will attempt to catch bad key names only if the package option tracingkeys is enabled.

You can add to the list of invalid key names by the following command:

New macros: \ltxkeys@badkeynames, \ltxkeys@addbadkeynames

```
\ltxkeys@badkeynames{⟨list⟩}
\ltxkeys@addbadkeynames{⟨list⟩}
```

where ⟨list⟩ is a comma-separated list of inadmissible names. The updating is done by merging, so that entries are not repeated in the internal list of bad key names.

You can remove from the list of bad key names by using the following command:

New macro: \ltxkeys@removebadkeynames

```
\ltxkeys@removebadkeynames{⟨list⟩}
```

where, again, ⟨list⟩ is comma-separated. It is not advisable to remove any member of the default bad key names.

11 Declaring options

New macros: \ltxkeys@declareoption, \ltxkeys@unknownoptionhandler

```
\ltxkeys@declareoption[⟨pref⟩]<⟨fam⟩>{⟨option⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@declareoption*[⟨pref⟩]<⟨fam⟩>{⟨cbk⟩}
\ltxkeys@unknownoptionhandler[⟨pref⟩]<⟨fam⟩>{⟨cbk⟩}
```

The unstarred variant of \ltxkeys@declareoption is simply a form of \ltxkeys@ordkey, with the difference that the key family ⟨fam⟩ is now optional and, when specified, must be given in angled brackets. The default family name is '\@currname.\@currext', i. e., the name of the class file or package and its file extension.

The starred (⋆) variant of \ltxkeys@declareoption prescribes the default action to be taken when undefined options with prefix ⟨pref⟩ and in family ⟨fam⟩ are passed to class or package. You may use \CurrentKey and \CurrentVal within this macro to pass the unknown option and its value to another class or package or to specify other actions. In fact, you can use #1 in this macro to represent the current key prefix, #2 for the current family, #3 for the current key name, and #4 for the value of the current key. The command \ltxkeys@unknownoptionhandler is equivalent to the starred (⋆) variant of \ltxkeys@declareoption.

Note 11.1 The starred (⋆) variant of \ltxkeys@declareoption differs from the starred form of LaTeX's \DeclareOption and the starred form of xkeyval package's \DeclareOptionX.

Examples: \ltxkeys@declareoption

```
554 \ltxkeys@declareoption*[KV]<mypackage>{%
555   \PackageWarning{mypackage}{%
556     Unknown option `\CurrentKey' with value `\InnocentVal' ignored}%
557 }

558 \ltxkeys@declareoption*{\PassOptionsToClass{#3}{article}}

559 \ltxkeys@unknownoptionhandler[KV]<mypackage>{%
560   \@expandtwoargs\in@{,#3,}{,\KV@mypackage@handledkeys,}%
561   \ifin@
562     % The reader may want to investigate what the parameter texts
563     % ##1 and ####1 below stand for:
564     \ltxkeys@ordkey[#1]{#2}{#3}[#4]{\def\x####1{####1xx##1}}%
565   \else
566     \PassOptionsToClass{#3}{myclass}%
567   \fi
568 }
```

See note 4.3 for the meaning of the parameter texts in this example. The contents of the macro \KV@mypackage@handledkeys are handled keys for key prefix KV and family fam. See section 8 for the meaning of handled keys.

New macros: \ltxkeys@declarecmdoption, \ltxkeys@declarebooloption, etc

```
569 \ltxkeys@declareordoption[⟨pref⟩]<⟨fam⟩>{⟨option⟩}[⟨dft⟩]{⟨cbk⟩}
570 \ltxkeys@declarecmdoption[⟨pref⟩]<⟨fam⟩>[⟨mp⟩]{⟨option⟩}[⟨dft⟩]{⟨cbk⟩}
571 \ltxkeys@declarebooloption[⟨pref⟩]<⟨fam⟩>[⟨mp⟩]{⟨option⟩}[⟨dft⟩]{⟨cbk⟩}
572 \ltxkeys@declarechoiceoption[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨option⟩}[⟨bin⟩]{⟨alt⟩}
573   [⟨dft⟩]{⟨cbk⟩}
```

These are the equivalents of the macros \ltxkeys@ordkey, \ltxkeys@cmdkey, \ltxkeys@boolkey and \ltxkeys@choicekey, respectively, but now the family ⟨fam⟩ is optional (as is ⟨pref⟩) and, when specified, must be given in angled brackets. The default family name for these new commands is '\@currname.\@currext', i. e., the current style or class filename and filename extension. \ltxkeys@declareordoption is equivalent to the unstarred variant of \ltxkeys@declareoption. See the choice keys in subsection 3.6 for the meaning of ⟨bin⟩ and ⟨alt⟩ associated with the command \ltxkeys@declarechoiceoption.

11.1 Options that share the same attributes

The commands

Macros

```
574 \ltxkeys@declareordoption
575 \ltxkeys@declarecmdoption
576 \ltxkeys@declarebooloption
577 \ltxkeys@declarechoiceoption
```

can each be used to introduce several options that share the same path or bases (option prefix, option family, and macro prefix) and callback ⟨cbk⟩. All that is needed is to replace ⟨option⟩ in

these commands with the comma-separated list ⟨options⟩. Because some users might prefer to see these commands in their plural forms when defining several options with the same callback, we have provided the following aliases.

New macros: \ltxkeys@declarecmdoptions, \ltxkeys@declarebooloptions, etc

```
\ltxkeys@declareordoptions[⟨pref⟩]<⟨fam⟩>{⟨option⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@declarecmdoptions[⟨pref⟩]<⟨fam⟩>[⟨mp⟩]{⟨option⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@declarebooloptions[⟨pref⟩]<⟨fam⟩>[⟨mp⟩]{⟨option⟩}[⟨dft⟩]{⟨cbk⟩}
\ltxkeys@declarechoiceoptions[⟨pref⟩]{⟨fam⟩}[⟨mp⟩]{⟨option⟩}[⟨bin⟩]{⟨alt⟩}
  [⟨dft⟩]{⟨cbk⟩}
```

11.2 Declaring all types of option with one command

New macro: \ltxkeys@declaremultitypeoptions

```
\ltxkeys@declaremultitypeoptions[⟨pref⟩]<⟨fam⟩>[⟨mp⟩]{%
  ⟨keytype⟩/⟨keyname⟩/⟨dft⟩/⟨cbk⟩;
  another set of key attributes;
  etc.
}
\ltxkeys@declaremultitypeoptions*[⟨pref⟩]<⟨fam⟩>[⟨mp⟩]{%
  ⟨keytype⟩/⟨keyname⟩/⟨dft⟩/⟨cbk⟩;
  another set of key attributes;
  etc.
}
```

Here, the key default value ⟨dft⟩ and callback ⟨cbk⟩ can be absent in all cases. ⟨keytype⟩ may be any one of {ord, cmd, sty, sty*, bool, choice}. The star (⋆) in 'sty⋆' has the same meaning as in \ltxkeys@stylekey above, namely, undefined dependants will be defined on the fly when the parent key is set. The optional quantity ⟨mp⟩ is the macro prefix, as in, for example, subsection 3.3. The syntax for the command \ltxkeys@declaremultitypeoptions is identical to that of \ltxkeys@declarekeys except for the following differences: For \ltxkeys@declarekeys the family is mandatory and must be given in curly braces, while for \ltxkeys@declaremultitypeoptions the family is optional, with the default value of '\@currname.\@currext', i. e., the name of the class file or package and its file extension. For \ltxkeys@declaremultitypeoptions, the optional family is expected to be given in angled brackets. The starred (⋆) variant of the command \ltxkeys@declaremultitypeoptions defines only undefined options. An alias for the long command \ltxkeys@declaremultitypeoptions is \declaremultitypeoptions.

Example: \ltxkeys@declaremultitypeoptions

```
\declaremultitypeoptions*[KV]<fam>[mp@]{%
  cmd/option1/xx/\def\x##1{##1xx#1};
  bool/option2/true;
}
```

12 Executing options

New macro: \ltxkeys@executeoptions

```
\ltxkeys@executeoptions[⟨prefs⟩]<⟨fams⟩>[⟨na⟩]{⟨keyval⟩}
```

This executes/sets the ⟨key⟩=⟨value⟩ pairs given in ⟨keyval⟩. The optional ⟨na⟩ specifies the list of keys (without values) to be ignored. ⟨prefs⟩ is the list of prefixes for the keys; and the optional ⟨fams⟩ signifies families in which the keys suggested in ⟨key⟩=⟨value⟩ have been defined. The default value of ⟨fams⟩ is \@currname.\@currext. The command \ltxkeys@executeoptions can thus be used to process keys with different prefixes and from several families.

13 Processing options

New macro: \ltxkeys@processoptions

```
598 \ltxkeys@processoptions[⟨prefs⟩]<⟨fams⟩>[⟨na⟩]
599 \ltxkeys@processoptions⋆[⟨prefs⟩]<⟨fams⟩>[⟨na⟩]
```

The command \ltxkeys@processoptions processes the ⟨key⟩=⟨value⟩ pairs passed by the user to the class or package. The optional argument ⟨na⟩ can be used to specify keys that should be ignored. The optional argument ⟨fams⟩ can be used to specify the families that have been used to define the keys. The default value of ⟨fams⟩ is \@currname.\@currext. The package command \ltxkeys@processoptions doesn't protect expandable macros in the user inputs unless the ltxkeys package is loaded before \documentclass, in which case it is also possible to use the command \XProcessOptions of the catoptions package. When used in a class file, the macro \ltxkeys@processoptions will ignore unknown keys or options. This allows the user to use global options in the \documentclass command which can be inherinted by packages loaded afterwards.

The starred (⋆) variant of \ltxkeys@processoptions works like the plain variant except that, if the ltxkeys package is loaded after \documentclass, it also copies user input from the command \documentclass. When the user specifies an option in the \documentclass which also exists in the local family or families of the package issuing \ltxkeys@processoptions⋆, the local key too will be set. In this case, #1 in the command \ltxkeys@declareoption (or a similar command) will be the value entered in the \documentclass command for this key. First the global options from \documentclass will set local keys and afterwards the local options, specified via \usepackage, \RequirePackage or \LoadClass, will set local keys, which could overwrite the previously set global options, depending on the way the options sections are constructed.

13.1 Hooks for 'before' and 'after' processing options

New macros: \ltxkeys@beforeprocessoptions, \ltxkeys@afterprocessoptions

```
600 \ltxkeys@beforeprocessoptions{⟨code⟩}
601 \ltxkeys@afterprocessoptions{⟨code⟩}
```

The macros \ltxkeys@beforeprocessoptions and \ltxkeys@afterprocessoptions can be used to process an arbitrary code given in ⟨code⟩ before and after \ltxkeys@processoptions has been executed. The command \ltxkeys@afterprocessoptions is particularly useful when it is required to optionally load a package, with the decision dependent on the state or outcome of an option in the current package. For obvious reasons, LaTeX's options parser doesn't permit the loading of packages in the options section. The command \ltxkeys@afterprocessoptions can be used to load packages after the current package's options have been processed. Here is an example for optionally loading some packages at the end of the options section:

Example: \ltxkeys@afterprocessoptions

```
602 \ltxkeys@cmdkey[KV]{fam}[mp@]{keya}[]{%
603   \iflacus#1\dolacus\else
604     \ltxkeys@afterprocessoptions{\RequirePackage[#1]{mypackage}}%
```

```
    \fi
}
```

In this example, #1 refers (as usual) to the user input for key keya. Here, we assume that the values of keya will be the ⟨key⟩=⟨value⟩ pairs for options of mypackage. The loading of mypackage will be determined by whether the user input for keya is empty or not. That is why keya has an empty default value. More complex application scenarios can, of course, be easily created[⋆8].

14 Key commands and key environments

Key commands and environments are commands and environments that expect ⟨key⟩=⟨value⟩ pairs as input, in addition to any number of possible nine conventional arguments. Key commands and environments have already been introduced by the keycommand and skeycommand packages, but the inherent robustness of the ltxkeys provides another opportunity to re-introduce these features here. The syntax here is also simpler and the new featureset has the following advantages over those in keycommand and skeycommand packages:

a) The defined commands and environments can have up to nine conventional parameters, in addition to the ⟨key⟩=⟨value⟩ pairs.
b) Anyone or all of the nine command or environment parameters can be delimited.
c) All the various types of key (command keys, boolean keys, etc.) can be used as the keys for the new command or environment.
d) With the prefixes \ltxkeysglobal and \ltxkeysprotected, global and robust key commands and environments can be defined in a manner that simulates TEX's \global and ε-TEX's \protected.
e) The exit code for the key environment can have access to the arguments of the environment, unlike in LATEX's environment.
f) Simple commands are provided for accessing the current values (and, in the case of boolean keys, the current states) of keys.

The specification of the mandatory arguments and any optional first argument for the key command and key environment has the same syntax as in LATEX's \newcommand and \newenvironment. The key command and key environment of the ltxkeys package have the syntaxes:

New macros: \ltxkeyscmd, \ltxkeysenv, etc

```
⟨pref⟩\ltxkeyscmd⟨cs⟩[⟨narg⟩][⟨dft⟩]<⟨delim⟩>(⟨keys⟩){⟨defn⟩}
⟨pref⟩\reltxkeyscmd⟨cs⟩[⟨narg⟩][⟨dft⟩]<⟨delim⟩>(⟨keys⟩){⟨defn⟩}
⟨pref⟩\ltxkeysenv{⟨name⟩}[⟨narg⟩][⟨dft⟩]<⟨delim⟩>(⟨keys⟩){⟨begdefn⟩}{⟨enddefn⟩}
⟨pref⟩\reltxkeysenv{⟨name⟩}[⟨narg⟩][⟨dft⟩]<⟨delim⟩>(⟨keys⟩){⟨begdefn⟩}{⟨enddefn⟩}
```

Here, ⟨pref⟩ is the optional command prefix, which may be either \ltxkeysglobal (for global commands) or \ltxkeysprotected (for ε-TEX protected commands); ⟨cs⟩ is the command; ⟨name⟩ is the environment name; ⟨narg⟩ is the number of parameters; ⟨dft⟩ is the default value of the first argument; ⟨delim⟩ are the parameter delimiters; ⟨keys⟩ are the keys to be defined for the command or environment; ⟨defn⟩ is the replacement text of the command; ⟨begdefn⟩ is the environment entry text; and ⟨enddefn⟩ is the code to execute while exiting the environment.

The ⟨keys⟩ have the same syntax as they do for the command \ltxkeys@declarekeys (subsection 3.8). The parameter delimiters ⟨delim⟩, given above in angled brackets, have the syntax:

⋆8 The command \iflacus, whose argument is delimited by \dolacus, tests for emptiness of its argument.

Parameter delimiters

```
611 1⟨delim1⟩ 2⟨delim2⟩ ... 9⟨delim9⟩
```

where ⟨delim1⟩ and ⟨delim2⟩ are the delimiters for the first and second parameters, respectively, etc. Only the parameters with delimiters are to be specified in ⟨delim⟩. Examples are provided later.

In the LaTeX \newenvironment and \renewenvironment commands, with the syntax

Macros: \newenvironment, \renewenvironment

```
612 \newenvironment{⟨name⟩}[⟨narg⟩][⟨dft⟩]{⟨begdefn⟩}{⟨enddefn⟩}
613 \renewenvironment{⟨name⟩}[⟨narg⟩][⟨dft⟩]{⟨begdefn⟩}{⟨enddefn⟩}
```

the environment's parameters and/or arguments aren't accessible in ⟨enddefn⟩. If the environment user wants to access the parameters in ⟨enddefn⟩, he has to save them while still in ⟨begdefn⟩. This isn't the case with the commands \ltxkeysenv and \reltxkeysenv, for which the user can access the environment parameters while in ⟨enddefn⟩. To do this, he should call the command \envarg, which expects as argument the corresponding numeral of the parameter text. For example, \envarg{1} and \envarg{3} refer to the first and third arguments of the environment, respectively. Examples are provided later. The current values of environment's keys can always be accessed in ⟨enddefn⟩.

But how do we access the current values or states of keys while in ⟨begdefn⟩ and ⟨enddefn⟩? To this end the commands \val, \ifval, \ifvalTF, \keyval, \ifkeyval and \ifkeyvalTF are provided. They have the following syntaxes:

New macros: \val, \ifval, \ifvalTF, etc

```
614 % The following commands don't first confirm that the key exists before
615 % attempting to obtain its current value or state. They are expandable:
616 \val{⟨key⟩}
617 \ifval⟨boolkey⟩\then ⟨true⟩ \else ⟨false⟩ \fi
618 \ifvalTF{⟨boolkey⟩}{⟨true⟩}{⟨false⟩}

619 % The following commands first confirm that the key exists before attempting
620 % to obtain its current value or state. They are expandable if the key
621 % is defined:
622 \keyval{⟨key⟩}
623 \ifkeyval⟨boolkey⟩\then ⟨true⟩ \else ⟨false⟩ \fi
624 \ifkeyvalTF{⟨boolkey⟩}{⟨true⟩}{⟨false⟩}
```

The command \val yields the current value of a command or environment key, irrespective of the type of key. Its argument should exclude the key-command name, key prefix, key family, and macro prefix. The command \ifval expects as argument a boolean key name ⟨boolkey⟩ (without the command name, key prefix, key family, and macro prefix) and yields either \iftrue or \iffalse. The command \ifvalTF expects as argument a boolean key and yields one of two LaTeX branches, ⟨true⟩ or ⟨false⟩.

The commands \val, \ifval and \ifvalTF can be used in expansion contexts (including in \csname...\endcsname) but if their arguments aren't defined as keys, they will return an undefined command, either immediately or later. On the hand, their counterparts (namely, the commands \keyval, \ifkeyval and \ifkeyvalTF) will first check that the key has been defined before attempting to obtain its current value or state. This affects their expandability when a key is undefined. My advice is that the user should always use \keyval, \ifkeyval and \ifkeyvalTF

instead of \val, \ifval and \ifvalTF, unless he is sure he hasn't committed any mistakes in key's name; but he might be writing a package—that contains these commands—for the use of the TEX community. Also, here there is an advantage in using \protected@edef in place of \edef: some LATEX commands are protected with \protect.

The commands \val, \ifval, \ifvalTF, \keyval, \ifkeyval and \ifkeyvalTF, like the command and environment keys, are available in ⟨defn⟩, ⟨begdefn⟩ and ⟨enddefn⟩. These commands (i. e., \val, \ifval, \ifvalTF, \keyval, \ifkeyval and \ifkeyvalTF) are pushed on entry into ⟨defn⟩ or ⟨begdefn⟩, and they are popped on exit of ⟨defn⟩ or ⟨enddefn⟩. Unless they're defined elsewhere outside the ltxkeys package, they're undefined outside ⟨defn⟩, ⟨begdefn⟩, ⟨enddefn⟩, and the environment body[★9].

14.1 Final tokens of every environment

The user can add some tokens to the very end of every subsequent environment by declaring those tokens in the macro \ltxkeys@everyeoe, which by default contains only LATEX's command \ignorespacesafterend. That is, the ltxkeys package automatically issues

Example: \ltxkeys@everyeoe

```
\ltxkeys@everyeoe{\ignorespacesafterend}
```

It is important to note that new tokens are prepended (and not appended) to the internal hook that underlies \ltxkeys@everyeoe, such that by default \ignorespacesafterend always comes last in the list. You can empty the list \ltxkeys@everyeoe by issuing \ltxkeys@everyeoe{} and rebuild it anew, still by prepending elements to it. \ltxkeys@everyeoe isn't actually a token list register but it behaves like one[†1]. It is safe to issue \ltxkeys@everyeoe{⟨token⟩} and/or \ltxkeys@everyeoe{} in the ⟨begdefn⟩ part of the key environment. One of the examples in subsection 14.2 illustrates this point.

Note 14.1 The pointer schemes of subsection 4.4 are applicable to key commands and key environments. The \needvalue pointer is used in one of the examples in subsection 14.2.

14.2 Examples of key command and environment

Examples: Key command

```
% It is possible to use parameter delimiters, as the following
% \@nil and \@mil show:
% \ltxkeysglobal\ltxkeysrobust\ltxkeyscmd*\cmdframebox
%    [3][default]<2\@nil 3\@mil>(⟨keys⟩){⟨defn⟩}

% No parameter delimiters for the following:
\ltxkeysglobal\ltxkeysrobust\ltxkeyscmd*\cmdframebox[3][default](%
  cmd/width/\textwidth;
  cmd/textcolor/black;
  cmd/framecolor/red;
  cmd/framerule/.4pt;
  cmd/framesep/4pt;
```

[★9] The commands \pathkeysval, \ifpathkeysval, \ifpathkeysvalTF, \pathkeyskeyval, \ifpathkeyskeyval and \ifpathkeyskeyvalTF are always available, but they can be used only in the context of 'pathkeys' (section 15).

[†1] However, you can't do \ltxkeys@everyeoe\expandafter{\cmd} because \ltxkeys@everyeoe isn't a token list

```
  bool/putframe/true;
  bool/testbool/true;
){%
  \begingroup
  \fboxrule\keyval{framerule}\relax
  \fboxsep\keyval{framesep}\relax
  \ifkeyval putframe\then
    \fcolorbox{\keyval{framecolor}}{gray!25}{%
  \fi
  \parbox{\keyval{width}}{%
    \color{\keyval{textcolor}}%
    Arg-1: #1\\
    Arg-2: #2\\
    Arg-3: #3%
  }%
  \ifkeyval putframe\then}\fi
  \ifkeyvalTF{testbool}{\def\x{T}}{\def\y{F}}%
  \endgroup
}

\begin{document}
\cmdframebox[Text-1]{Text-2\\ ...\\ text-3}{Text-4}(%
  width=.5\textwidth,
  framecolor=cyan,
  textcolor=purple,
  framerule=1pt,
  framesep=10pt,
  putframe=true
)
\end{document}
```

Example: Key environment

```
\ltxkeysenv*{testenv}[1][right](%
  cmd/xwidth/2cm;
  cmd/ywidth/1.5cm;
  cmd/body;
  cmd/\needvalue{author}/\null;
  bool/boola/false;
){%
  \ltxkeys@iffound{,#1,}\in{,right,left,}\then\else
    \@latex@error{Unknown text alignment type ‘#1’}\@ehd
  \fi
  \centering
  \fbox{\parbox{\keyval{xwidth}}{\usename{ragged#1}\keyval{body}}}%
  \ifkeyval boola\then\color{red}\fi
  \fbox{\parbox{\keyval{ywidth}}{\usename{ragged#1}\keyval{body}}}%
  \normalcolor
  % \val, \ifval, etc, are unavailable in \ltxkeys@everyeoe. Hence
```

register.

```
  % we save the value of 'author' here:
  \protected@edef\quoteauthor{\val{author}}%
  % Re-initialize \ltxkeys@everyeoe:
  \ltxkeys@everyeoe{}%
  \ltxkeys@everyeoe{\ignorespacesafterend}%
  \ltxkeys@everyeoe{\endgraf\vskip\baselineskip
    \centerline{\itshape\quoteauthor}}
  % Just to test parameter use inside \ltxkeysenv:
  \def\testmacroa##1{aaa##1}%
}{%
  \def\testmacrob##1{##1bbb}%
}

\begin{document}
\begin{testenv}(%
  xwidth=5cm,
  ywidth=4cm,
  boola=true,
  author={Cornelius Tacitus \textup{(55--120~AD)}},
  body={Love of fame is the last thing even learned men can bear
    to be parted from.}
)
\end{testenv}
\end{document}
```

Examples: Key environment

```
% The following line has parameter delimiters \@nil  and \@mil:
% \ltxkeysglobal\ltxkeysrobust\ltxkeysenv*{envframebox}
%    [3][default]<2\@nil 3\@mil>(⟨defn⟩){}

% No parameter delimiters for the following:
\ltxkeysglobal\ltxkeysrobust\ltxkeysenv*{envframebox}[3][default](%
  cmd/width/\textwidth/\def\xx##1{##1};
  cmd/textcolor/black;
  cmd/framerule/.4pt;
  ord/framecolor/brown;
  bool/putframe/true;
){%
  \begingroup
  \fboxrule\val{framerule}\relax
  \ifval putframe\then\fcolorbox{\val{framecolor}}{gray!25}{\fi
  \parbox{\val{width}}{%
    Arg-1: #1\\
    Arg-2: \textcolor{\val{textcolor}}{#2}\\
    Arg-3: #3%
  }%
  \ifval putframe\then}\fi
  \endgroup
}{%
  \edef\firstarg{\envarg{1}}%
  \def\yy##1{##1}%
}
```

```
\begin{document}
\begin{envframebox}[Text-1]{Text-2\\ ...\\ test text-2}{Text-3}(%
  width=.5\textwidth,
  textcolor=purple,
  framerule=1pt,
  putframe=true
)
\end{envframebox}
\end{document}
```

Examples: Nested key environments

```
\def\testenv{}
\reltxkeysenv{testenv}(%
  % The \y below is just a test:
  cmd/fraclen/0.1cm/\def\y##1{#1yyy##1};
  cmd/framerule/.4pt;
  cmd/framecolor/blue;
  cmd/textcolor/black;
  bool/putframe/true;
){%
  \cptdimdef\tempb{.5\textwidth-\val{fraclen}*\currentgrouplevel}%
  \noindent
  \endgraf\fboxrule=\val{framerule}\relax
  \color{\val{framecolor}}%
}{}

\begin{document}
\begin{testenv}(%
  fraclen=0.1cm,
  framerule=1.5pt,
  framecolor=red,
  textcolor=magenta,
  putframe=true
)%
\ifval putframe\then\fbox{\fi
\parbox\tempb{%
  \color{\val{textcolor}}%
  outer box\endgraf
  ***aaa***
  \vspace*{5mm}%
  \begin{testenv}(%
    fraclen=0.1cm,
    framerule=3pt,
    framecolor=green,
    textcolor=cyan,
    putframe=true
  )%
  \ifval putframe\then\fbox{\fi
  \parbox\tempb{%
    \color{\val{textcolor}}%
    inner box\endgraf\vspace*{5mm}%
```

```
    +++bbb+++
  }%
  \ifval putframe\then}\fi
  \end{testenv}%
}%
\ifval putframe\then}\fi
\end{testenv}
\end{document}
```

The following example shows that in place of the functions \val, \ifval, \ifvalTF, \keyval, \ifkeyval and \ifkeyvalTF the user can access the values and states of keys by concatenating the command or environment name, the '@' sign and the name of the key. This, of course, requires that '@' has catcode 11.

Examples: Key command

```
\ltxkeyscmd\myframebox[2][default text](%
  cmd/width/\textwidth;
  cmd/textcolor/black;
  cmd/framecolor/black;
  cmd/framesep/3\p@;
  cmd/framerule/0.4\p@;
  % The following is choice key 'textalign' with default value 'center'.
  % The '.code=' in the admissible values is optional, but not the forward
  % slash '/'.
  choice/textalign.{%
      center/.code=\def\ttextalign{center},
      left/.code=\def\ttextalign{flushleft},
      right/.code=\def\ttextalign{flushright}
    }/center;
  bool/putframe/true
){%
  \begingroup
  \fboxsep\myframebox@framesep
  \fboxrule\myframebox@framerule\relax
  \cptdimdef\myframebox@boxwidth
    {\myframebox@width-2\fboxsep-2\fboxrule}%
  \noindent\begin{lrbox}\@tempboxa
  \begin{minipage}[c][\height][s]\myframebox@boxwidth
  \@killglue
  \begin\ttextalign
  \textcolor{\myframebox@textcolor}{Arg-1: #1\endgraf Arg-2: #2}%
  \end\ttextalign
  \end{minipage}%
  \end{lrbox}%
  \@killglue
  \color{\myframebox@framecolor}%
  \ifmyframebox@putframe\fbox{\fi
    \usebox\@tempboxa
  \ifmyframebox@putframe}\fi
  \endgroup
}

\begin{document}
```

```
\myframebox[Text-1]{Test text-2\\ ...\\test text-2}
  (framerule=2pt,framecolor=blue,textcolor=purple,
  putframe=true,textalign=right)
\end{document}
```

15 Pathkeys

Let us start this section with a welcome message: you don't have to repeatedly type in long key paths and commands when using pathkeys. There is help ahead on how to reduce estate when using pathkeys.

The pathkeys package can be loaded on its own (via \RequirePackage or \usepackage) or as an option to the ltxkeys package (see Table 1). All the options listed in Table 1 are accepted by the pathkeys package. They are all passed on to ltxkeys package, except pathkeys that is simply ignored by pathkeys package.

Pathkeys are keys with a tree or directory structure[†2]. When defining and setting pathkeys, the full key path is usually required. This is also the case when seeking the current value or state of a key. When using pathkeys the user is relieved of the need to known and remember where the optional arguments have to be placed in calls to macros. And like the commands \ltxkeys@definekeys and \ltxkeys@declarekeys, pathkeys are automatically initialized after definition, i. e., they are automatically set with their default values.

The command for defining and setting pathkeys is \pathkeys, which has the following syntax. The same command is used for several other tasks related to pathkeys. The 'flag' entry in the argument of \pathkeys determines the action that the command is expected to take.

New macros: \pathkeys

```
\pathkeys{⟨main⟩/⟨sub⟩/⟨subsub⟩/.../⟨flag⟩: ⟨attrib⟩}
```

In the argument of command \pathkeys, the 'path' is made up of '⟨main⟩/⟨sub⟩/⟨subsub⟩/.../'. The quantity ⟨main⟩ is the main path and ⟨sub⟩ is the sub path, etc. Note that there is no forward slash (/) before ⟨main⟩. If the path is empty, the default path 'dft@main/dft@sub/', or the user-supplied current path, is used. There is more about the default and current paths later in this guide.

The ⟨attrib⟩ is determined by the key called ⟨flag⟩. The ⟨flag⟩ must be a member of the set described in Table 2. See the table notes for the ⟨attrib⟩'s of the flags. The attributes describe the arguments associated with the flags, i. e., the quantities expected after the colon ':' in the argument of \pathkeys. The ⟨na⟩ is the list of keys that are ignored. If it is present in the attribute ⟨attrib⟩ part of \pathkeys, it must always be given in square brackets '[]' (see note 15.1).

For flags with star sign (⋆) the user should make sure there is no space between the flag and its star.

Table 2: Flags and attributes for pathkeys

No.	Flag	Meaning
1	define	Define the keys whether or not they already exist. See note 2.1

Continued on next page

[†2] This might sound like pgf keys, but the semantics, syntaxes, and the implementation here are all different from

Continued from last page

No.	Flag	Meaning
2	`define*`	Define the keys only if they don't already exist.[2.2]
3	`declareoptions`	Declare the given options whether or not they already exist.[2.3]
4	`declareoptions*`	Declare the options if they don't already exist.[2.4]
5	`set`	Set the listed keys.[2.5]
6	`executeoptions`	Execute the listed options.[2.6]
7	`processoptions`	Process the listed options in the order in which they were declared, and don't copy `\documentclass` options.[2.7]
8	`processoptions*`	Process the listed options in the order in which they appear in the command `\usepackage`, and copy `\documentclass` options.[2.8]
9	`launch`	Launch the listed keys (see subsection 4.8).[2.9]
10	`store value`	Store the value of ⟨`key`⟩ in the given ⟨`macro`⟩.[2.10]
11	`print value`	Print the current value of ⟨`key`⟩.[2.11]
12	`add value`	Add the specified value to the current value of key.[2.12]
13	`ifbool`	Test the state of a boolean key. This returns ⟨`true`⟩ or ⟨`false`⟩.[2.13]
14	`ifdef`	Test if key is currently defined on the given single path. This returns ⟨`true`⟩ or ⟨`false`⟩.[2.14]
15	`ifkeyonpath`	Test if key is currently defined on any of the given comma-separated multiple paths. This returns ⟨`true`⟩ or ⟨`false`⟩.[2.15]
16	`disable`	Immediately disable the given keys.[2.16]
17	`disable*`	Disable the given keys at the hook `\AtBeginDocument` and not immediately.[2.17]
18	`key handler` or `handler`	Unknown key handler.[2.18]
19	`option handler`	Unknown option handler (see subsection 4.9). Options are keys with a special default family. There might be a reason to handle unknown options separately from unknown keys.

Table 2 notes

These notes describe the attributes of handlers, i. e., what are required to be specified in `\pathkeys` command after the colon '`:`' sign. ⟨`na`⟩ keys must appear in square brackets, e. g., `[keya]`.

[2.1]See attribute in note 15.1. [2.2]Same as for `define` flag. [2.3]Same as for `define` flag. [2.4]The flag `declareoptions*` simply signifies the user's aim to define definable options; it has nothing to do with the starred (⋆) variant of the command `\ltxkeys@declareoption` of section 11. Attribute: same as for `define` flag. [2.5]⟨`na`⟩ keys and ⟨`key`⟩=⟨`value`⟩ pairs (see section 4). [2.6]⟨`na`⟩ keys and ⟨`key`⟩=⟨`value`⟩ pairs (see section 12). [2.7]⟨`na`⟩ keys (see section 13). [2.8]⟨`na`⟩ keys (see section 13). [2.9]⟨`key`⟩=⟨`value`⟩ pairs.[2.10]⟨`key`⟩ and ⟨`macro`⟩.[2.11]⟨`key`⟩.[2.12]⟨`key`⟩ and ⟨`value`⟩.[2.13]⟨`key`⟩.[2.14]⟨`key`⟩.[2.15]⟨`key`⟩.[2.16]The attribute is a comma-separated key list.[2.17]Comma-separated key list.[2.18]The arguments of the unknown key or option handler are the main path, subpaths (separated by forward slash), key name, and the current key value (see subsection 4.9). The key or option handler can have up to a maximum of 4 arguments.

Note 15.1 The syntax for specifying keys to be defined by `\pathkeys` is (see subsection 3.8)

Syntax for defining keys in \pathkeys

```
\pathkeys{⟨path⟩/define:
  ⟨keytype⟩/⟨keyname⟩/⟨dft⟩/⟨cbk⟩;
  another set of key attributes;
  etc.
}
```

The default value ⟨`dft`⟩ and the callback ⟨`cbk`⟩ can be absent in all cases. ⟨`keytype`⟩ may be any member of the set `{ord, cmd, sty, sty*, bool, choice}`. The star (⋆) in 'sty⋆' has the same

those of `pgf` keys.

meaning as in \ltxkeys@stylekey (subsection 3.4), namely, undefined dependants will be defined on the fly when the parent is set/executed.

Example: Syntax for defining pathkeys

```
\pathkeys{fam/subfam/subsubfam/define:
  cmd/keya/defaultval/\def\cmda#1{#1};
  bool/keyb/true;
}
```

Choice keys must have their names associated with their nominations (i. e., admissible values) in the format ⟨keyname⟩.⟨{nominations}⟩, as below (see also subsection 3.8):

Syntax for defining choice keys in \pathkeys

```
% 'keya' is a choice key with simple nominations and callback, while 'keyb'
% is a choice key with complex nominations. The function \order is generated
% internally by the package for choice keys.
\pathkeys{fam/subfam/subsubfam/define:
  choice/keya.{left,right,center}/center/
    \edef\x{\ifcase\order 0\or 1\or 2\fi};
  choice/keyb.{%
    center/.code=\def\textalign{center},
    left/.code=\def\textalign{flushleft},
    % '.code=' can be omitted, as in:
    right/\def\textalign{flushright},
    justified/\let\textalign\relax
  }/center/\def\x##1{##1xx#1};
}
```

The ⟨na⟩ keys, if they are present in the attribute of \pathkeys, must always be given in square brackets []. They can come either before or after the ⟨key⟩=⟨value⟩ list to be set in the current run. For example,

Example: 'na' keys

```
\pathkeys{fam/subfam/subsubfam/define:
  cmd/keya/xx/\def\cmda#1{#1};
  bool/keyb/true;
}
% Set 'keya' and ignore 'keyb':
\pathkeys{fam/subfam/subsubfam/set: keya=zz,keyb=true [keyb]}
% or
\pathkeys{fam/subfam/subsubfam/set: [keyb] keya=zz,keyb=true}
```

See subsection 15.4 for further examples of the use of ignored keys. Here we can see that a value is provided for 'keyb' and yet we're ignoring the key. However, in practical applications it is often impossible to predict the set of keys (among a set of them) that may be executed at any time by the user of the keys. Therefore, ⟨na⟩ keys are much more useful than the above example demonstrates.•

Some of the commands associated with pathkeys are listed below. The abbreviation ⟨pk⟩ means the full key path and key name, all separated by forward slash.

New macros: \pathkeysval, \ifpathkeysval, \ifpathkeysvalTF, etc.

```
% The following commands are expandable:
\pathkeysval{⟨pk⟩}
\ifpathkeysval{⟨pk⟩} \then ... \else ... \fi
\ifpathkeysvalTF{⟨pk⟩}{⟨true⟩}{⟨false⟩}
% The following commands aren't expandable:
\pathkeyskeyval{⟨pk⟩}
\ifpathkeyskeyval{⟨pk⟩} \then ... \else ... \fi
\ifpathkeyskeyvalTF{⟨pk⟩}{⟨true⟩}{⟨false⟩}
\pathkeys@storevalue{⟨pk⟩}⟨cmd⟩
```

The commands \pathkeysval and \pathkeyskeyval simply yield the current value of the key. The commands \ifpathkeysval and \ifpathkeyskeyval, which require \then to form balanced conditionals, test the current state of the boolean key ⟨pk⟩ in a TEX-like syntax. The commands \ifpathkeysvalTF and \ifpathkeyskeyvalTF also test the current state of the boolean key ⟨pk⟩ but return ⟨true⟩ or ⟨false⟩ in a LATEX syntax. The command \pathkeys@storevalue stores the current value of key ⟨pk⟩ in the given command ⟨cmd⟩.

Note 15.2 If called outside an assignment or document environment, the macros \pathkeysval and \pathkeyskeyval can give 'no document error', to signify that a token has been output outside these situations. And one source of problem with \ifpathkeysval and \ifpathkeyskeyval is to omit \then after their argument. If you find yourself typing long key paths and the commands \pathkeysval and \pathkeyskeyval, etc., repeatedly, there is help ahead on how to reduce estate when using pathkeys.

The following provide our first examples of pathkeys and a demonstration of some of the commands associated with pathkeys.

Examples: Pathkeys

```
\pathkeys{fam/subfam/subsubfam/define:
  cmd/xwidth/\@tempdima/\def\y##1{#1yy##1};
  cmd/keya/\def\cmda#1{#1};
  bool/putframe/true;
}
\pathkeys{fam/subfam/subsubfam/set: putframe=true [keya]}
\pathkeys{fam/subfam/subsubfam/ifdef: xwidth}{\def\x{T}}{\def\x{F}}
\pathkeys{fam/subfam/subsubfam,famx/subfamx/subsubfamx/ifkeyonpath: xwidth}
  {\def\x{T}}{\def\x{F}}
\pathkeys{fam/subfam/subsubfam/print value: xwidth}=\z@pt
\pathkeys{fam/subfam/subsubfam/store value: keya \cmd}
\pathkeys{fam/subfam/subsubfam/add value: keya=\def\cmdb#1{#1}}
\pathkeys@storevalue{fam/subfam/subsubfam/putframe}\cmd
\edef\x{\ifpathkeysvalTF{fam/subfam/subsubfam/putframe}{T}{F}}
\edef\x{\ifpathkeysval fam/subfam/subsubfam/putframe\then T\else F\fi}
\edef\x{\ifpathkeysval fam/subfam/subsubfam/putframe\then T\else F\fi}
% 'xputframe' is undefined. What does the following return?
\edef\x{\pathkeysval{fam/subfam/subsubfam/xputframe}}
% Unknown key handler:
\pathkeys{fam/subfam/subsubfam/key handler:
  % '#1' is the key's main path, '#2' is the subpaths combined,
  % '#3' is the key name, and '#4' is the current value of the key:
  \ltxkeys@warn{Unknown key '#3' with value '#4' ignored.}%
```

```
}
\pathkeys{fam/subfam/subsubfam/disable*: keya,keyb,keyc}
```

Examples: Pathkeys

```
\pathkeys{KV/frame/framebox/define*:
  cmd/width/\textwidth/\def\x##1{#1xx##1};
  cmd/textcolor/black;
  cmd/framecolor/black;
  cmd/framesep/3\p@;
  cmd/framerule/0.4\p@;
  cmd/cornersize/20\p@;
  choice/textalign.{%
      center/.code=\def\ttextalign{center},
      left/.code=\def\ttextalign{flushleft},
      right/.code=\def\ttextalign{flushright}
    }/center;
  bool/putframe/true;
  cmd/arga;
  cmd/argb;
}

\newcommand*\myframebox[1][]{%
% Use 'set' or 'launch' here, but they don't have the same meaning:
  \pathkeys{KV/frame/framebox/set:#1}%
  \begingroup
  \fboxsep\pathkeysval{KV/frame/framebox/framesep}%
  \fboxrule\pathkeysval{KV/frame/framebox/framerule}\relax
  \cptdimdef\boxwidtha{%
    \pathkeysval{KV/frame/framebox/width}-2\fboxsep-2\fboxrule
  }%
  \noindent\begin{lrbox}\@tempboxa
  \begin{minipage}[c][\height][s]\boxwidtha
  \@killglue
  \begin\ttextalign
  \textcolor{\pathkeysval{KV/frame/framebox/textcolor}}{%
    Arg-1: \pathkeysval{KV/frame/framebox/arga}
    \endgraf
    Arg-2: \pathkeysval{KV/frame/framebox/argb}%
  }%
  \end\ttextalign
  \end{minipage}%
  \end{lrbox}%
  \@killglue
  \color{\pathkeysval{KV/frame/framebox/framecolor}}%
  \ifpathkeysval{KV/frame/framebox/putframe}\then\ovalbox{\fi
    \usebox\@tempboxa
  \ifpathkeysval{KV/frame/framebox/putframe}\then}\fi
  \endgroup
}

\begin{document}
\myframebox[arga=Text-1,argb={Test text-2\\ ...\\test text-2},
```

```
939   framerule=2pt,framecolor=blue,textcolor=purple,
940   putframe=true,textalign=right]
941 \end{document}
```

Note 15.3 When using pathkeys (and in general the commands `\ltxkeys@definekeys` and `\ltxkeys@declarekeys`), there is a potential problem in deploying forward slashes in key defaults and macros without enclosing those slashes in curly braces. They will confuse the parser. Several solutions exist, including tweaking the relevant internal parser, but I haven't decided on the optimal solution to this possibility. For example, the following will fail:

Example: Forward slashes in key defaults and macros

```
942 \pathkeys{fam/subfam/subsubfam/define*:
943   bool/keya/true/\ifpathkeysval fam/subfam/subsubfam/keya\then
944     \def\x{T}\else\def\x{F}\fi;
945 }
```

Its correct form is

Example: Forward slashes in key defaults and macros

```
946 \pathkeys{fam/subfam/subsubfam/define*:
947   bool/keya/true/\ifpathkeysval{fam/subfam/subsubfam/keya}\then
948     \def\x{T}\else\def\x{F}\fi;
949 }
```

15.1 Shortened pathkeys commands

As seen above, the estate for deploying pathkeys can be large when compared with the amount of typing required for conventional keys presented in the previous chapters. To reduce the estate, the first line of thought is to store any long path in a macro and call the macro instead of the path. The path is always fully expanded under safe actives. The following example demonstrates this approach.

Examples: Putting paths in macros

```
950 \def\mypath{fam/subfam/subsubfam}
951 \pathkeys{\mypath/define:
952   cmd/xwidth/\@tempdima/\def\y##1{#1yy##1};
953   cmd/keya/\def\cmda#1{#1};
954   bool/putframe/true
955 }
956 \pathkeys{famx/subfamx,fam/subfam/ifkeyonpath: xwidth}{\def\x{T}}{\def\x{F}}
957 \pathkeys{famx/subfamx,\mypath/ifkeyonpath: xwidth}{\def\x{T}}{\def\x{F}}
958 \pathkeys{\mypath/set: putframe=true}
959 \pathkeys{\mypath/ifdef: xwidth}{\def\x{T}}{\def\x{F}}
960 \pathkeys{\mypath/print value: xwidth}=\z@pt
961 \pathkeys@storevalue{\mypath/putframe}\cmd
962 \edef\x{\ifpathkeysvalTF{\mypath/putframe}{T}{F}}
963 \edef\x{\ifpathkeysval \mypath/putframe\then T\else F\fi}
964 \edef\x{\ifpathkeysval \mypath/putframe\then T\else F\fi}
965 \pathkeys{\mypath/add value: keya=\def\cmdb#1{#1}}
```

Instead of defining your own commands like the above \mypath, you can use the following namespaced commands:

New macros: \newpath, \defpath, \changepath, \undefpath, \usepath

```
\newpath{⟨pathname⟩}{⟨path⟩}
\defpath{⟨pathname⟩}{⟨path⟩}
\changepath{⟨pathname⟩}{⟨path⟩}
\undefpath{⟨pathname⟩}
\usepath{⟨pathname⟩}
```

These commands have their own separate namespace. Here, ⟨pathname⟩ is used, after definition, as an abbreviation for the full path ⟨path⟩. The command \newpath creates ⟨pathname⟩ if it didn't already exist; \defpath creates ⟨pathname⟩ whether or not it exists; \changepath is equivalent to \defpath; \undefpath undefines ⟨pathname⟩; and \usepath expands ⟨pathname⟩ to its full meaning. The macro \usepath does accept, as argument, a comma-separated list of pathnames, but it is unlikely that such a list will, for now anyway, be needed outside when using the flag ifkeyonpath (see the following example).

Examples: \newpath, \usepath

```
\newpath{path1}{fam/subfam/subsubfam1}
\newpath{path2}{fam/subfam/subsubfam2}
\pathkeys{\usepath{path1}/define:
  cmd/keya/xx/\def\cmda#1{#1};
  bool/keyb/true
}
\pathkeys{\usepath{path1,path2}/ifkeyonpath: keya}{\def\x{T}}{\def\x{F}}
\pathkeys@storevalue{\usepath{path1}/keyb}\cmd
\edef\x{\ifpathkeysvalTF{\usepath{path1}/keya}{T}{F}}
```

The following shortened counterparts of the pathkeys commands are provided (see Table 3). The abbreviated commands are available only after the user has invoked \pathkeys@useshortcmds, which expects no argument. The command \pathkeys@useshortcmds has only local effect, i. e., the abbreviations may be localized to a group. The abbreviations are defined only if they're definable (i. e., didn't exist before calling the command \pathkeys@useshortcmds).

Table 3: Pathkeys command abbreviations

Command	Abbreviation	Command	Abbreviation
\pathkeysval	\pkv	\pathkeyskeyval	\pkkv
\ifpathkeysval	\ifpkv	\ifpathkeyskeyval	\ifpkkv
\ifpathkeysvalTF	\ifpkvTF	\ifpathkeyskeyvalTF	\ifpkkvTF

The user isn't constrained to use the short form commands of Table 3. He/she can define his/her own short forms by using the command \pathkeys@makeshortcmds, which has the syntax:

New macro: \pathkeys@makeshortcmds

```
\pathkeys@makeshortcmds{⟨short-1⟩=⟨long-1⟩, ..., ⟨short-n⟩=⟨long-n⟩}
```

where ⟨short-i⟩ and ⟨long-i⟩ are the short (new) and long (existing) aliases of the command ⟨i⟩. The equality sign (=) is mandatory here. You don't have to (in fact, you shouldn't) call \pathkeys@useshortcmds after calling \pathkeys@makeshortcmds.

Example: \pathkeys@makeshortcmds

```
\pathkeys@makeshortcmds{\kval=\pathkeyskeyval,\ifkvalTF=\ifpathkeyskeyvalTF}
```

15.2 Default and current paths

New macros: \pathkeys@currentpath, etc.

```
\pathkeys@addtodefaultpath{⟨path⟩}
\pathkeys@changedefaultpath{⟨path⟩}
\pathkeys@currentpath{⟨path⟩}
\pathkeys@usedefaultpath
\pathkeys@pushcurrentpath
\pathkeys@popcurrentpath
\pathkeys@pathhistory
```

If the key path is empty, then the current path will be used; and if there is no current path, the default path will be used. The default path is `dft@main/dft@sub`. This can be changed by the commands \pathkeys@addtodefaultpath and \pathkeys@changedefaultpath. The current path can be declared by providing an argument to the command \pathkeys@currentpath. The default path can be made the current path by invoking the command \pathkeys@usedefaultpath, which is parameterless.

It isn't mandatory, but it is useful, to first push the prevailing path before changing it. This can be done by calling the parameterless command \pathkeys@pushcurrentpath. When you're done with the current path, you can revert to the path before the current path by calling the command \pathkeys@popcurrentpath. You can get the entire history of path changes from the container \pathkeys@pathhistory, which is useful in complex situations.

Before the current path is resorted to, the path for the commands \pathkeys, \pathkeysval, \ifpathkeysval, etc. must be empty (i. e., no main and no subs). Therefore, in any given setting, the path that is dominant can be made current so that it isn't given in \pathkeys, \pathkeysval, \ifpathkeysval, etc. The non-dominant paths could then be listed in full. Of course, there can't be more than one current path.

Examples: \pathkeys@currentpath, etc.

```
\newcommand*\myframebox[1][]{%
  \pathkeys@currentpath{KV/frame/framebox}%
  \pathkeys{launch:#1}%
  \begingroup
  \pathkeys@useshortcmds
  \fboxsep\pkv{framesep}\fboxrule\pkv{framerule}\relax
  \cptdimdef\boxwidtha{\pkv{width}-2\fboxsep-2\fboxrule}%
  \noindent\begin{lrbox}\@tempboxa
  \begin{minipage}[c][\height][s]\boxwidtha
  \@killglue
  \begin\ttextalign
  \textcolor{\pkv{textcolor}}{Arg-1: \pkv{arga}\endgraf Arg-2: \pkv{argb}}%
  \end\ttextalign
  \end{minipage}%
  \end{lrbox}%
  \@killglue
  \color{\pkv{framecolor}}%
  \ifpkv{putframe}\then\ovalbox{\fi
```

```
    \usebox\@tempboxa
  \ifpkv{putframe}\then}\fi
  \endgroup
}

\begin{document}
\myframebox[arga=Text-1,argb={Test text-2\\ ...\\test text-2},
  framerule=2pt,framecolor=blue,textcolor=purple,
  putframe=true,textalign=right]
\end{document}
```

15.3 Nested pathkeys

The command \pathkeys can be nested, as the following example shows:

Example: Nested pathkeys

```
\def\mypath{fam/subfam/subsubfam}
\pathkeys{\mypath/define:
  cmd/xwidth/\@tempdima/\def\y##1{#1yy##1};
  % The default, not callback, of 'keya' is \def\cmda#1{#1}. The key
  % has no callback:
  cmd/keya/\def\cmda#1{#1};
  % The callback of 'keyb' says ''if 'keyb' is 'true', define 'keyc''':
  bool/keyb/true/
    \pathkeys{\mypath/ifbool: keyb}{%
      \pathkeys{\mypath/define: cmd/keyc/xx/\def\cmdc####1{####1#1}}%
    }{%
      % 'keyd' has no callback:
      \pathkeys{\mypath/define: choice/keyd.{yes,no}/yes}%
    }
}
\pathkeys{\mypath/set: keyb=true}
```

Try to find out why the following produces an error:

Example: Nested pathkeys

```
\def\mypath{fam/subfam/subsubfam}
\pathkeys{\mypath/define:
  cmd/keya/keyadefault/
    \pathkeys{\mypath/define*: cmd/keyb/xx/\def\cmdb####1{####1}};
}
\pathkeys{\mypath/set: keya=bbb}
```

The reason is that keyb was defined when the default was being set up for keya after the definition of keya. The second setting of keya prompts an error that keyb is being redefined. Notice that keyb is to be defined uniquely by the flag define*. To avoid this type of error, you may consider removing * from define*.

15.4 Pathkeys as class or package options

To use the command \pathkeys for declaring class or package options, the user should simply call \pathkeys with the flag declareoptions (or declareoptions* for defining only unique options).

The flags executeoptions, processoptions and processoptions* can be used to execute and process options, respectively. In this respect, although not necessary, you may want to change the default or current path to reflect the class file or package name.

Example: Declaring and processing options

```
\ProvidesPackage{mypackage}[2011/11/11 v0.1 My test package]
\newpath{mypath}{mypackage/myfunc/myfunckeys}
% Declare three unique options:
\pathkeys{\usepath{mypath}/declareoptions*:
  cmd/opt1/12cm/\def\y##1{#1yy##1};
  bool/opt2/true/\ifpathkeysval{\usepath{mypath}/opt2}\then
    \def\x{T}\else\def\x{F}\fi;
  ord/opt3/zz/\def\z##1{#1zz##1};
}
% Set up defaults for options 'opt1' and 'opt2', ignoring option 'opt3':
\pathkeys{\usepath{mypath}/executeoptions:
  opt1=10cm,opt2=true,opt3=yy [opt3]
}
% Ignore 'opt1' when processing options:
\pathkeys{\usepath{mypath}/processoptions*: [opt1]}

\documentclass[opt1=2cm,opt2=false]{article}
\usepackage[opt3=somevalue]{mypackage}
```

16 Some miscellaneous commands

Some of the macros used internally by the ltxkeys package are available to the user. A few of them are described below.

16.1 Trimming leading and trailing spaces

New macros: \ltxkeys@trimspaces, \ltxkeys@trimspacesincs

```
\ltxkeys@trimspaces{⟨token⟩}⟨cs⟩
\ltxkeys@trimspacesincs⟨cs⟩
```

The command \ltxkeys@trimspaces trims (i. e., removes) the leading and trailing spaces around ⟨token⟩ and returns the result in the macro ⟨cs⟩. Forced (i. e., explicit) leading and trailing spaces around ⟨token⟩ are removed unless they are enclosed in braces.

The command \ltxkeys@trimspacesincs trims the leading and trailing spaces around the token in the macro ⟨cs⟩ and returns the result in ⟨cs⟩.

16.2 Checking user inputs

New macros: \ltxkeys@checkchoice, \ltxkeys@checkinput, \CheckUserInput

```
\ltxkeys@checkchoice[⟨parser⟩](⟨val⟩⟨order⟩){⟨input⟩}{⟨nomin⟩}{⟨true⟩}
\ltxkeys@checkchoice*[⟨parser⟩](⟨val⟩⟨order⟩){⟨input⟩}{⟨nomin⟩}{⟨true⟩}
\ltxkeys@checkchoice+[⟨parser⟩](⟨val⟩⟨order⟩){⟨input⟩}{⟨nomin⟩}{⟨true⟩}{⟨false⟩}
\ltxkeys@checkchoice*+[⟨parser⟩](⟨val⟩⟨order⟩){⟨input⟩}{⟨nomin⟩}{⟨true⟩}{⟨false⟩}
```

```
\ltxkeys@checkinput{⟨input⟩}{⟨nomin⟩}{⟨true⟩}{⟨false⟩}
\CheckUserInput{⟨input⟩}{⟨nomin⟩}
```

The command \ltxkeys@checkchoice is a re-implementation of xkeyval package's command \XKV@checkchoice so as to accept arbitrary list parser ⟨parser⟩ and more robustness. It checks the user input ⟨input⟩ against the list of nominations ⟨nomin⟩. If the input is valid, the user input is returned in ⟨val⟩ and the numerical order (starting from zero) of the input in the nominations is returned in ⟨order⟩[†3]. If the input isn't valid, the user input is still returned in ⟨val⟩, but −1 is returned in ⟨order⟩. ⟨parser⟩ is the list parser. The starred (⋆) variant of \ltxkeys@checkchoice will convert ⟨input⟩ into lowercase before checking it against the nominations. The plus (+) variant of \ltxkeys@checkchoice expects two branches (⟨true⟩ and ⟨false⟩) of callback at the end of the test. The non-plus variant expects only one branch (⟨true⟩) and will return error if the input is invalid[†4].

The commands \ltxkeys@checkinput and \CheckUserInput apply to comma-separated lists of nominations ⟨nomin⟩ and they always convert ⟨input⟩ to lowercase before checking it against the nominations ⟨nomin⟩. The macro \ltxkeys@checkinput expects two branches of callback, while \CheckUserInput expects no callback. Instead, \CheckUserInput will toggle the internal boolean \ifinputvalid to true if the input is valid, and to false otherwise. The internal boolean \ifinputvalid could then be called by the user after the test.

16.3 Does a test string exist in a string?

New macros: \ltxkeys@in, \ltxkeys@iffound

```
\ltxkeys@in{⟨teststr⟩}{⟨str⟩}
\ltxkeys@in*{⟨teststr⟩}{⟨str⟩}{⟨true⟩}{⟨false⟩}
\ltxkeys@iffound⟨teststr⟩\in⟨str⟩\then ⟨true⟩ \else ⟨false⟩ \fi
```

The unstarred variant of the command \ltxkeys@in is identical with LaTeX2ε kernel's (2011/06/27) \in@. The command \in@ tests for the presence of ⟨teststr⟩ in ⟨str⟩. The starred (⋆) variant of \ltxkeys@in returns two LaTeX branches ⟨true⟩ and ⟨false⟩. On the other hand, the command \ltxkeys@iffound requires the first argument to be delimited by \in and the second argument by \then.

Example: \ltxkeys@iffound

```
\ltxkeys@iffound xx\in aax\then \def\x{T}\else \def\x{F}\fi
```

Note 16.1 The command \ltxkeys@iffound trims leading and trailing spaces around the tokens ⟨teststr⟩ and ⟨str⟩ before the test! The commands \ltxkeys@in and \ltxkeys@iffound aren't expandable.

16.4 Does a given pattern exist in the meaning of a macro?

New macro: \ltxkeys@ifpattern

```
\ltxkeys@ifpattern{⟨teststr⟩}⟨cmd⟩{⟨true⟩}{⟨false⟩}
```

[†3] The functions ⟨val⟩ and ⟨order⟩ are user-supplied macros.
[†4] There is also \ltxkeys@commacheckchoice, whose parser is implicitly comma ',' and does not need to be given

The command \ltxkeys@ifpattern simply determines if the meaning of ⟨cmd⟩ contains ⟨teststr⟩. It returns ⟨true⟩ if ⟨teststr⟩ is found in the meaning of ⟨cmd⟩, and ⟨false⟩ otherwise.

16.5 \ifcase for arbitrary strings

New macros: \ltxkeys@ifcase, \ltxkeys@findmatch

```
\ltxkeys@ifcase{⟨teststr⟩}{⟨case-1⟩:⟨cbk-1⟩,...,⟨case-n⟩:⟨cbk-n⟩}
  {⟨true⟩}{⟨false⟩}
\ltxkeys@findmatch{⟨teststr⟩}{⟨case-1⟩:⟨cbk-1⟩,...,⟨case-n⟩:⟨cbk-n⟩}{⟨fn⟩}
```

The command \ltxkeys@ifcase tests ⟨teststr⟩ against ⟨case-i⟩. If a match is found, the ⟨case-i⟩'s callback ⟨cbk-i⟩ is returned in the macro \currmatch and ⟨true⟩ is executed. If at the end of the loop no match is found, \ltxkeys@ifcase returns empty \currmatch and executes ⟨false⟩.

The command \ltxkeys@findmatch works like \ltxkeys@ifcase but executes the fallback ⟨fn⟩ (instead of ⟨true⟩ or ⟨false⟩) when no match is found.

16.6 Is the number of elements from a sublist found in a csv list $\geq n$?

New macro: \ltxkeys@ifincsvlistTF

```
\ltxkeys@ifincsvlistTF[Aparser](⟨nr⟩){⟨sub⟩}{⟨main⟩}{⟨true⟩}{⟨false⟩}
\ltxkeys@ifincsvlistTF*[Aparser](⟨nr⟩){⟨sub⟩}{⟨main⟩}{⟨true⟩}{⟨false⟩}
```

The command \ltxkeys@ifincsvlistTF checks if the number of elements of ⟨parser⟩-separated (csv) list ⟨sub⟩ found in ⟨main⟩ is equal or greater than ⟨nr⟩. The argument ⟨main⟩ is the main list and ⟨sub⟩ is the sublist of test strings. Normally, ⟨sub⟩ will be a user input and ⟨main⟩ the list of nominations. Neither ⟨main⟩ nor ⟨sub⟩ is expanded in the test. If the test is true, \ltxkeys@itemspresent returns all the elements found, \ltxkeys@nritems returns the number of elements found, and ⟨true⟩ is executed. If the test fails, \ltxkeys@itemspresent returns empty, \ltxkeys@nritems returns −1, and ⟨false⟩ is executed. The starred (⋆) variant of \ltxkeys@ifincsvlistTF will turn both input and nominations to lowercase before the test. The default values of the optional list ⟨parser⟩ and the optional number of elements to find ⟨nr⟩ are comma ',' and 1, respectively.

16.7 Is the number of elements from a sublist found in a tsv list $\geq n$?

New macro: \ltxkeys@ifintsvlistTF

```
\ltxkeys@ifintsvlistTF(⟨nr⟩){⟨sub⟩}{⟨main⟩}{⟨true⟩}{⟨false⟩}
\ltxkeys@ifintsvlistTF*(⟨nr⟩){⟨sub⟩}{⟨main⟩}{⟨true⟩}{⟨false⟩}
```

The command \ltxkeys@ifintsvlistTF checks if the number of elements of nonparser-separated (tsv) list ⟨sub⟩ found in ⟨main⟩ is equal or greater than ⟨nr⟩. The argument ⟨main⟩ is the main list and ⟨sub⟩ is the sublist of test strings. Normally, ⟨sub⟩ will be a user input and ⟨main⟩ the list of nominations. Neither ⟨main⟩ nor ⟨sub⟩ is expanded in the test. If the test is true,

by the user.

\ltxkeys@itemspresent returns all the elements found, \ltxkeys@nritems returns the number of elements found, and ⟨true⟩ is executed. If the test fails, \ltxkeys@itemspresent returns empty, \ltxkeys@nritems returns −1, and ⟨false⟩ is executed. The starred (⋆) variant of \ltxkeys@ifintsvlistTF will turn both input and nominations to lowercase before the test.

Normally, tsv-matching requires that the test strings in ⟨sub⟩ are unique in the nominations ⟨main⟩. Some caution is, therefore, necessary when dealing with tsv lists.

16.8 Is the number of elements in a csv list $\geq n$ or $\leq n$?

New macro: \ltxkeys@ifeltcountTF

```
\ltxkeys@ifeltcountTF[⟨parser⟩](⟨rel⟩){⟨nr⟩}{⟨list⟩}{⟨true⟩}{⟨false⟩}
\ltxkeys@ifeltcountTF⋆[⟨parser⟩](⟨rel⟩){⟨nr⟩}{⟨listcmd⟩}{⟨true⟩}{⟨false⟩}
```

The command \ltxkeys@ifeltcountTF checks if the number of elements in ⟨parser⟩-separated list ⟨list⟩ has relation ⟨rel⟩ (>=<) with number ⟨nr⟩. If the test is true, ⟨true⟩ is executed, otherwise ⟨false⟩ is executed. The starred (⋆) variant of \ltxkeys@ifeltcountTF will expand ⟨listcmd⟩ once before the test. Double parsers and empty entries in ⟨list⟩ are ignored. The default values of the optional list ⟨parser⟩ and the optional relational type ⟨rel⟩ are comma ',' and '=', respectively. The number ⟨nr⟩ is a mandatory argument.

The following example returns ⟨false⟩ (i. e., \meaning\x -> F).

Example: \ltxkeys@ifeltcountTF

```
\ltxkeys@ifeltcountTF[;](<){2}{a;b;c}{\def\x{T}}{\def\x{F}}
```

16.9 What is the numerical order of an element in a csv list?

New macro: \ltxkeys@getorder

```
\ltxkeys@getorder[⟨parser⟩]{⟨elt⟩}{⟨list⟩}
\ltxkeys@getorder⋆[⟨parser⟩]{⟨elt⟩}{⟨listcmd⟩}
```

The command \ltxkeys@getorder returns in \ltxkeys@order the numerical order of ⟨elt⟩ in ⟨parser⟩-separated ⟨list⟩ or ⟨listcmd⟩. The value of \ltxkeys@order is the numerical order of the first match found. The count starts from zero (0). The starred (⋆) variant will expand ⟨listcmd⟩ once before commencing the search for ⟨elt⟩. If no match is found, \ltxkeys@order returns −1, which can be used for taking further decisions.

16.10 List normalization

New macros: \ltxkeys@commanormalize, \ltxkeys@kvnormalize

```
\ltxkeys@commanormalize{⟨list⟩}⟨cmd⟩
\ltxkeys@commanormalizeset{{⟨list-1⟩}⟨cmd-1⟩,...,{⟨list-n⟩}⟨cmd-n⟩}
\ltxkeys@kvnormalize{⟨list⟩}⟨cmd⟩
\ltxkeys@kvnormalizeset{{⟨list-1⟩}⟨cmd-1⟩,...,{⟨list-n⟩}⟨cmd-n⟩}
```

These commands will normalize the comma-separated ⟨list⟩ (or ⟨list-i⟩) and return the result in ⟨cmd⟩ (or ⟨cmd-i⟩). For the command \ltxkeys@kvnormalize, ⟨list⟩ is assumed to be a list of ⟨key⟩=⟨value⟩ pairs. Normalization implies changing the category codes of all the active commas to their standard values, as well as trimming leading and trailing spaces around the elements of

the list and removing consecutive multiple commas. Thus empty entries that are not enforced by curly braces are removed. Besides dealing with multiple commas and the spaces between entries, the command \ltxkeys@kvnormalize removes spaces between keys and the equality sign, and multiple equality signs are made only one. Further, the category codes of comma and the equality sign are made normal throughout the list.

16.11 Parsing arbitrary csv or kv list

New macro: \ltxkeys@parse

1084 \ltxkeys@parse⟨flag⟩[⟨parser⟩]{⟨list⟩}
1085 \ltxkeys@parse⋆⟨flag⟩[⟨parser⟩]{⟨listcmd⟩}

The unexpandable command \ltxkeys@parse is the list processor for the ltxkeys package. It can process both arbitrary ⟨parser⟩-separated lists and ⟨key⟩=⟨value⟩ pairs. The ⟨flag⟩, which must lie in the range (0, 3), determines the type of processing that is required. The admissible values of ⟨flag⟩ and their meaning are given in Table 4. The macro \ltxkeys@parse loops over the given ⟨parser⟩-separated ⟨list⟩ and execute the user-defined, one-parameter command \ltxkeys@do for every item in the list, passing the item as an argument and preserving outer braces. The default value of ⟨parser⟩ is comma ',' . The starred (⋆) variant of \ltxkeys@parse will expand ⟨listcmd⟩ once before commencing the loop.

Table 4: Flags for command \ltxkeys@parse

Flag	Meaning
0	⟨list⟩ is assumed to be an ordinary list (i. e., not a list of ⟨key⟩=⟨value⟩ pairs); it isn't normalized by \ltxkeys@parse prior to parsing.
1	⟨list⟩ is assumed to be an ordinary list (i. e., not a list of ⟨key⟩=⟨value⟩ pairs); it is normalized by \ltxkeys@parse prior to parsing.
2	⟨list⟩ is assumed to be a list of ⟨key⟩=⟨value⟩ pairs; it isn't normalized by \ltxkeys@parse prior to parsing.
3	⟨list⟩ is assumed to be a list of ⟨key⟩=⟨value⟩ pairs; it is normalized by \ltxkeys@parse prior to parsing.

Here are some points to note about the list processor \ltxkeys@parse:

a) If an item contains ⟨parser⟩, it must be wrapped in curly braces when using \ltxkeys@parse, otherwise the elements may be mixed up during parsing. The braces will persist thereafter, but will of course be removed during printing (if the items are printed).
b) White spaces before and after the list separator are always ignored by the normalizer called by \ltxkeys@parse. If an item contains ⟨parser⟩ or starts with a space, it must, therefore, be wrapped in curly braces before calling \ltxkeys@parse.
c) Since when ⟨flag⟩ is 0 or 2 the command \ltxkeys@parse doesn't call the normalizer, in this case it does preserve outer/surrounding spaces in the entries. Empty entries in ⟨list⟩ or ⟨listcmd⟩ will be processed by \ltxkeys@parse if the boolean \ifltxkeys@useempty is true. You may thus issue the command \ltxkeys@useemptytrue before calling \ltxkeys@parse. The ability to parse empty entries is required by packages that use empty key prefixes, and/or families[†5]. \ifltxkeys@useempty is false by default.
d) The command \ltxkeys@parse can be nested to any level and can be mixed with other

[†5] The use of empty key prefixes, families and paths is, in general, not advisable.

looping macros.

e) In the command \ltxkeys@parse, it is always possible to break out of the loop prematurely at any level of nesting, simply by issuing the command \loopbreak. Breaking an inner loop doesn't affect the continuation of the outer loop, and vice versa.

f) The argument of the one-parameter command \ltxkeys@do can be passed to a multi-parameter command, or to a command that expects delimited arguments.

16.12 Expandable list parser

New macro: \ltxkeys@declarelistparser

```
\ltxkeys@declarelistparser⟨iterator⟩{⟨parser⟩}
\def⟨processor⟩#1{...#1...}
⟨iterator⟩{⟨list⟩}⟨processor⟩
⟨iterator⟩!{⟨list⟩}⟨processor⟩
```

Given a parser (or list separator) ⟨parser⟩, the command \ltxkeys@declarelistparser can be used to define an expandable list iterator ⟨iterator⟩. The item processor ⟨processor⟩ should be a one-parameter macro, which will receive and process each element of ⟨list⟩. The optional exclamation mark (!) determines whether or not the processor is actually expanded and executed in the current expansion context. If ! is given, the processor is expanded and executed, otherwise it is merely given the elements as argument without expansion. In general, ⟨list⟩ isn't normalized, but is expanded once, before commencing the loop. The list can be normalized by the command \csv@@normalize (see catoptions package) before looping. The following example demonstrates the concept. The user can insert \listbreak as an item in the list to break out of the iteration prematurely.

Examples: \ltxkeys@declarelistparser

```
\ltxkeys@declarelistparser\iterator{;}
\def\do#1{#1}
% The following example will yield '\x=macro:->\do{a}\do{b}\do{c}':
\edef\x{\iterator{a;b;c}\do}
% The following example will yield '\x=macro:->abc':
\edef\x{\iterator!{a;b;c}\do}

% The following example will add 'a,b,c' to macro \y:
\ltxkeys@declarelistparser\doloop{,}
\doloop{a,b,c}{\cptaddtolist\y}
% The following example will add 'd,e' to macro \y:
\doloop!{d,e,\listbreak,f}{\cptaddtolist\y}
```

16.13 Remove one or all occurrences of elements from a csv list

New macro: \ltxkeys@removeelements

```
\ltxkeys@removeelements[⟨parser⟩](⟨nr⟩)⟨listcmd⟩{⟨sublist⟩}{⟨fd⟩}{⟨nf⟩}
\ltxkeys@removeelements*[⟨parser⟩](⟨nr⟩)⟨listcmd⟩{⟨sublist⟩}{⟨fd⟩}{⟨nf⟩}
```

The command \ltxkeys@removeelements removes ⟨nr⟩ number of each element of ⟨sublist⟩ from ⟨listcmd⟩. The default values of the optional list ⟨parser⟩ and the optional maximum number of elements to remove ⟨nr⟩ are comma ',' and 1, respectively. If at least one member of ⟨sublist⟩ is found and removed from ⟨listcmd⟩, then the callback ⟨fd⟩ is returned and

executed, otherwise ⟨nf⟩ is returned. Both ⟨fd⟩ and ⟨nf⟩ provide some fallback following the execution of \ltxkeys@removeelements. The challenge to the user is to remember that the command \ltxkeys@removeelements requires these callbacks, which may both be empty. The starred (⋆) variant of \ltxkeys@removeelements will remove from ⟨listcmd⟩ all the members of ⟨sublist⟩ found irrespective of the value of ⟨nr⟩. The optional ⟨nr⟩ is therefore redundant when the starred (⋆) variant of \ltxkeys@removeelements is called. Here, ⟨sublist⟩ is simply ⟨parser⟩-separated.

Example: \ltxkeys@removeelements

```
\def\xx{a;b;c;d;d;e;f;c;d}
% Remove at most 2 occurrences of ‘c’ and ‘d’ from \xx:
\ltxkeys@removeelements[;](2)\xx{c;d}{\def\x{done}}{\def\x{nil found}}
% Remove all occurrences of ‘c’ and ‘d’ from \xx:
\ltxkeys@removeelements*[;]\xx{c;d}{\def\x{done}}{\def\x{nil found}}
```

16.14 Replace one or all occurrences of elements in a csv list

New macro: \ltxkeys@replaceelements

```
\ltxkeys@replaceelements[⟨parser⟩](⟨nr⟩)⟨listcmd⟩{⟨sublist⟩}{⟨fd⟩}{⟨nf⟩}
\ltxkeys@replaceelements*[⟨parser⟩](⟨nr⟩)⟨listcmd⟩{⟨sublist⟩}{⟨fd⟩}{⟨nf⟩}
```

The command \ltxkeys@replaceelements replaces ⟨nr⟩ number of each element of ⟨sublist⟩ in ⟨listcmd⟩. The default values of the optional list ⟨parser⟩ and the optional maximum number of elements to replace ⟨nr⟩ are comma ‘,’ and 1, respectively. If at least one member of ⟨sublist⟩ is found and replaced in ⟨listcmd⟩, then the callback ⟨fd⟩ is returned and executed, otherwise ⟨nf⟩ is returned. Both ⟨fd⟩ and ⟨nf⟩ provide some fallback following the execution of \ltxkeys@replaceelements. The challenge to the user is to remember that the command \ltxkeys@replaceelements requires these callbacks, which may both be empty. The starred (⋆) variant of \ltxkeys@replaceelements will replace in ⟨listcmd⟩ all the members of ⟨sublist⟩ found irrespective of the value of ⟨nr⟩. The optional ⟨nr⟩ is therefore redundant when the starred (⋆) variant of \ltxkeys@replaceelements is used. Here, the syntax of ⟨sublist⟩ is as follows:

Sublist for \ltxkeys@replaceelements

```
{{⟨old-1⟩}{⟨new-1⟩}⟨parser⟩...⟨parser⟩{⟨old-n⟩}{⟨new-n⟩}}
```

where ⟨old-i⟩ is the element to be replaced and ⟨new-i⟩ is its replacement.

Example: \ltxkeys@replaceelements

```
\def\xx{a;b;c;d;d;e;f;c;d}
% Replace at most 2 occurrences of ‘c’ and ‘d’ in \xx with ‘s’ and ‘t’,
% respectively:
\ltxkeys@replaceelements[;](2)\xx{c{s};d{t}}{\def\x{done}}{\def\x{nil found}}
% Replace all occurrences of ‘c’ and ‘d’ in \xx with ‘s’ and ‘t’:
\ltxkeys@replaceelements*[;]\xx{c{s};d{t}}{\def\x{done}}{\def\x{nil found}}
```

16.15 Stripping outer braces

The list and key parsers of the ltxkeys package preserve outer braces. But sometimes it is needed to rid a token of one or more of its outer braces. This can be achieved by the following commands:

New macros: \ltxkeys@stripouterbraces, \ltxstripouterbraces

```
\ltxkeys@stripouterbraces⟨nr⟩{⟨token⟩}
\ltxstripouterbraces⟨nr⟩{⟨token⟩}⟨cmd⟩
```

The command \ltxkeys@stripouterbraces strips ⟨nr⟩ number of outer braces from ⟨token⟩. The command \ltxstripouterbraces strips ⟨nr⟩ number of outer braces from ⟨token⟩ and returns the result in the macro ⟨cmd⟩. The command \ltxkeys@stripouterbraces is expandable, but \ltxstripouterbraces isn't expandable. Normally, ⟨token⟩ shouldn't be expanded by these commands after the outer braces have been stripped off.

Examples: \ltxkeys@stripouterbraces, \ltxstripouterbraces

```
\toks@\expandafter\expandafter\expandafter
  {\ltxkeys@stripouterbraces{2}{{{\y}}}}
\edef\x{\unexpanded\expandafter\expandafter\expandafter
  {\ltxkeys@stripouterbraces{\@m}{{{\y}}}}}%
\ltxstripouterbraces{2}{{{{\y}}}}\x
```

If it is required to strip all the outer braces off ⟨token⟩, then ⟨nr⟩ can be made large, e. g., \@m.

17 To-do list

This section details additional package features that may become available in the foreseeable future. User views are being solicited in regard of the following proposals.

17.1 Patching key macros

Patching the macro of an existing key, instead of redefining the key. etoolbox package's \patchcmd doesn't permit the patching of commands with nested parameters. But since key macros may have nested parameters, a new patching scheme is to be first explored.

17.2 Modifying the dependant keys of an existing style key

New macros: \ltxkeys@adddepkeys, etc

```
\ltxkeys@adddepkeys[⟨pref⟩]{⟨fam⟩}{⟨paren⟩}{⟨deps⟩}
\ltxkeys@removedepkeys[pref]{fam}{⟨paren⟩}{⟨deps⟩}
\ltxkeys@replacedepkeys[pref]{fam}{⟨paren⟩}{⟨olddeps⟩}{⟨newdeps⟩}
```

Here, ⟨paren⟩ is the parent key of dependants keys; ⟨deps⟩ is the full specification of new or existing dependant keys (as in subsection 3.4), with their default values and callbacks; ⟨olddeps⟩ are the old dependants to replace with ⟨newdeps⟩. This would require patching macros of the form \⟨pref⟩@⟨fam⟩@⟨key⟩@dependants, which might have nested parametered-commands.

17.3 Toggle and switch keys

Introduce toggle keys and switch keys, but in practice who needs them? I would really need user views before implementing this feature. Toggles and switches, found in, e. g., the catoptions package, are more efficient than conventional booleans in the sense that each of them introduces and requires only one command, while each native boolean defines and requires up to three commands. However, toggles and switches haven't yet made it into the popular imagination of TEX users.

18 Version history

The following change history highlights significant changes that affect user utilities and interfaces; changes of technical nature are not documented in this section. The star sign (⋆) on the right-hand side of the following lists means the subject features in the package but is not reflected anywhere in this user guide.

Version 0.0.2 [2011/09/01]

'Pathkeys' introduced . . . . . section 15

User guide completed. . . . . ⋆

Version 0.0.1 [2011/07/30]

First public release. . . . . ⋆

Index

Index numbers refer to page numbers.