The keys3 package*
Key management for LaTeX3

Joseph Wright†

2009/06/12

1 Key management

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. For the user, the system normally results in input of the form

```
\PackageControlMacro{
  key-one = value one,
  key-two = value two
}
```

or

```
\PackageMacro[
  key-one = value one,
  key-two = value two
]{argument}.
```

For the programmer, the original keyval package gives only the most basic interface for this work. All key macros have to be created one at a time, and as a result the kvoptions and xkeyval packages have been written to extend the ease of creating keys. A very different approach has been provided by the pgfkeys package, which uses a key–value list to generate keys.

The keys3 package is aimed at creating a programming interface for key–value controls in LaTeX3. Keys are created using a key–value interface, in a similar manner to pgfkeys. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { module }
  key-one .code:n = code including parameter #1,
  key-two .set    = \l_module_store_tl
}
```

*This file has version number 110, last revised 2009/06/12.

†E-mail: joseph.wright@morningstar2.co.uk

These values can then be set as with other key–value approaches:

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

At a document level, the `\keys_set:nn` macro used within a document function. For LaTeX 2_ε, a generic set up function could be created with

```
\newcommand*\SomePackageSetup[1]{%
  \@nameuse{keys_set:nn}{module}{#1}%
}
```

or to use key–value input as the optional argument for a macro:

```
\newcommand*\SomePackageMacro[2][]{%
  \begingroup
    \@nameuse{keys_set:nn}{module}{#1}%
    % Main code for \SomePackageMacro
  \endgroup
}
```

The same concepts using xparse for LaTeX3 use:

```
\DeclareDocumentCommand \SomePackageSetup { m } {
  \keys_set:nn { module } { #1 }
}
\DeclareDocumentCommand \SomePackageMacro { o m } {
  \group_begin:
    \keys_set:nn { module } { #1 }
    % Main code for \SomePackageMacro
  \group_end:
}
```

Key names may contain any tokens, as they are handled internally using `\tl_to_str:n`. As will be discussed in section 1.2, it is suggested that the character "`/`" is reserved for sub-division of keys into logical groups. Macros are *not* expanded when creating key names, and so:

```
\tl_set:Nn \l_module_tmp_tl { key }
\keys_define:nn { module } {
  \l_module_tmp_tl .code:n = code
}
```

will create a key called `\l_module_tmp_tl`, and not one called `key`.

1.1 Creating keys

`\keys_define:nn` `\keys_define:nn` ⟨*module*⟩ ⟨*keyval list*⟩
Parses the ⟨*keyval list*⟩ and defines the keys listed there for ⟨*module*⟩. This function is designed for use in code, and therefore does not check the category codes of characters or ignore spaces.

Setting up and altering keys is carried out using one or more properties. The properties determine how a key acts, and may require zero, one or two argument: this is indicated by an argument specifier, in the same way as a standard LaTeX3 function. If only a single argument is required, braces around `n` arguments can be omitted.[1]

`.choice:` ⟨*key*⟩ `.choice:`
Sets ⟨*key*⟩ to act as a multiple choice key. Creating choices is discussed in section 1.3.

`.code:n` `.code:x` ⟨*key*⟩ `.code:n =` ⟨*code*⟩
Stores the ⟨*code*⟩ for execution when ⟨*key*⟩ is called. The ⟨*code*⟩ can include one parameter (`#1`).

`.code:Nn` `.code:Nx` ⟨*key*⟩ `.code:Nn =` ⟨*number*⟩ ⟨*code*⟩
Stores the ⟨*code*⟩ for execution when ⟨*key*⟩ is called. The ⟨*code*⟩ can include ⟨*number*⟩ parameters, which can be in the standard TeX range 0–9.

`.default:n` `.default:V` ⟨*key*⟩ `.default:n =` ⟨*content*⟩
Creates a default value for ⟨*key*⟩, which is used if no value is given.

TeXhackers note: The ⟨*content*⟩ is stored as a token list variable, and this means that there are some restrictions on the nature of ⟨*content*⟩.

`.generate_choices:nn` `.generate_choices:nx` ⟨*key*⟩ `.generate_choices:nn =` ⟨*comma list*⟩ ⟨*code*⟩

Makes ⟨*key*⟩ a multiple choice key, accepting the choices specified in ⟨*comma list*⟩. Each choice will execute ⟨*code*⟩ if it given. Within ⟨*code*⟩, the name of the current choice is available as `\l_keys_choice_tl`, and its position in the ⟨*comma list*⟩ as `\l_keys_choice_int`. Multiple choices are discussed further in section 1.3.

`.set:N` `.set_x:N` ⟨*key*⟩ `.set:N =` ⟨*variable*⟩
Defines ⟨*key*⟩ to store the value given in ⟨*variable*⟩. The type and scope (local or global)

[1] This is a general feature of key–value input methods.

of ⟨*variable*⟩ are determined from the name. The `x` version performs an expanded assignment.

TEXhackers note: A `\⟨variable⟩_set:Nn` function must exist to allow setting of the ⟨*variable*⟩.

`.set_bool:N` ⟨*key*⟩ `.set_bool:N =` ⟨*bool*⟩
`.set_bool_inverse:N` ⟨*key*⟩ `.set_bool_inverse:N =` ⟨*bool*⟩

Defines ⟨*key*⟩ to set ⟨*bool*⟩ to ⟨*value*⟩ (which must be either `true` or `false`). The `inverse` version sets the switch to the opposite logical sense to the argument given.

`.value_forbidden:`
`.value_required:` ⟨*key*⟩ `.value_forbidden:`

Flags for forbidding and requiring a ⟨*value*⟩ for ⟨*key*⟩.

1.2 Sub-dividing keys

When creating large numbers of keys, it may be desirable to divide them into several sub-groups for a given module. This can be achieved either by adding a sub-division to the module name:

```
\keys_define:nn { module / subgroup } {
  key .code:n = code
}
```

or to the key name:

```
\keys_define:nn { module } {
  subgroup / key .code:n = code
}
```

As illustrated, the best choice of token for sub-dividing keys in this way is "`/`". This is because of the method that is used to represent keys internally. Both of the above code fragments set the same key, which has full name `module/subgroup/key`.

As will be illustrated in the next section, this subdivision is particularly relevant to making multiple choices.

1.3 Multiple choices

In keys3, multiple choices are created by setting the `.choice:` property:

```
\keys_define:nn { module } {
  key .choice:
}
```

For keys which are set up as choices, the valid choices are generated by creating sub-keys of the choice key. This can be carried out in two ways.

In many cases, choices execute similar code which is dependant only on the name of the choice or the position of the choice in the list of choices. Here, the keys can share the same code, and can be rapidly created using the `.generate_choices:nn` property:

```
\keys_define:nn { module } {
  key .generate_choices:nn = {
    choice-a, choice-b, choice-c
  } {
    You~gave~choice~``\l_keys_choice_tl'',~
    which~is~in~position~\l_keys_choice_int
    \~in~the~list.
  }
}
```

`\l_keys_choice_tl`
`\l_keys_choice_int`
Inside the code block, the variables `\l_keys_choice_tl` and `\l_keys_choice_int` are available to indicate the name of the current choice, and its position in the comma list. The position is indexed from 1.

On the other hand, it is sometimes useful to create choices which use entirely different code from one another. This can be achieved by setting the `.choice:` property of a key, then manually defining sub-keys.

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

It is possible to mix the two methods, but manually-created choices should *not* use `\l_keys_choice_tl` or `\l_keys_choice_int`.

1.4 Setting keys

`\keys_set:nn` `\keys_set:nn` ⟨*module*⟩ ⟨*keyval list*⟩
Parses the ⟨*keyval list*⟩, and sets those keys which are defined for ⟨*module*⟩. The behaviour on finding an unknown key can be set by defining a special `unknown` key: this will be illustrated later. In contrast to `\keys_define:nn`, this function does check category codes and ignore spaces, and is therefore suitable for user input.

If a key is not known, `\keys_set:nn` will look for a special `unknown` key for the same module. This mechanism can be used to create new keys from user input.

```
\keys_define:nn { module } {
  unknown .code:n =
    You~tried~to~set~key~`\l_keys_path_tl'~to~`#1'
}
```

`\l_keys_key_tl` When processing an unknown key, the name of the key is available as `\l_keys_key_tl`. Note that this will have been processed using `\tl_to_str:N`. The value passed to the key (if any) is available as the macro parameter `#1`.

1.5 Examining keys: internal representation

`\keys_show:nn` `\keys_show:nn` ⟨*module*⟩ ⟨*key*⟩
Shows the internal representation of a ⟨*key*⟩. The function which executes a ⟨*key*⟩ is called `\keys >` ⟨*module*⟩`/`⟨*key*⟩`.cmd:w`.

1.6 Internal functions

`\keys_bool_set:N`
`\keys_bool_set_inverse:N` `\keys_bool_set:N` ⟨*bool*⟩

Creates code to set ⟨*bool*⟩ when ⟨*key*⟩ is given.

`\keys_choice_make:` `\keys_choice_make:`
Makes ⟨*key*⟩ a choice key.

`\keys_choices_generate:nx` `\keys_choices_generate:nx` ⟨*comma list*⟩ ⟨*code*⟩

Makes ⟨*comma list*⟩ choices for ⟨*key*⟩, each using ⟨*code*⟩.

`\keys_choice_find:n` `\keys_choice_find:n` ⟨*choice*⟩
Searches for ⟨*choice*⟩ as a sub-key of ⟨*key*⟩.

`\keys_cmd_set:nNn`
`\keys_cmd_set:nNx` `\keys_cmd_set:nNn` ⟨*path*⟩ ⟨*num args*⟩ ⟨*code*⟩
Creates a function for ⟨*path*⟩, taking ⟨*num args*⟩ and using ⟨*code*⟩.

`\keys_default_set:n`
`\keys_default_set:V` `\keys_default_set:n` ⟨*default*⟩
Sets ⟨*default*⟩ for ⟨*key*⟩.

`\keys_define_elt:n`
`\keys_define_elt:nn` `\keys_define_elt:n` ⟨*key*⟩ ⟨*value*⟩
Processing functions for key–value pairs when defining keys.

`\keys_define_key:n` `\keys_define_key:n` ⟨*key*⟩
Defines ⟨*key*⟩.

`\keys_execute:` `\keys_execute:`
Executes ⟨*key*⟩.

`\keys_execute_unknown:` `\keys_execute_unknown:`

Handles unknown ⟨*key*⟩ names.

`\keys_if_value_requirement:nTF` `\keys_if_value_requirement:nTF` ⟨*requirement*⟩ ⟨*true code*⟩ ⟨*false code*⟩

Check if ⟨*requirement*⟩ applies to ⟨*key*⟩.

`\keys_property_find:n` `\keys_property_find:n` ⟨*key*⟩

Separates ⟨*key*⟩ from ⟨*property*⟩.

`\keys_property_new:nn` `\keys_property_new:nn` ⟨*property*⟩ ⟨*code*⟩

Makes a new ⟨*property*⟩ expanding to ⟨*code*⟩

`\keys_property_undefine:n` `\keys_property_undefine:n` ⟨*property*⟩

Deletes ⟨*property*⟩ of ⟨*key*⟩.

`\keys_set_elt:n` `\keys_set_elt:nn` `\keys_set_elt:n` ⟨*key*⟩ ⟨*value*⟩
Processing functions for key–value pairs when setting keys.

`\keys_tmp:w` `\keys_tmp:w` ⟨*args*⟩
Used to store ⟨*code*⟩ to execute a ⟨*key*⟩.

`\keys_value_or_default:n` `\keys_value_or_default:n` ⟨*value*⟩

Sets `\l_keys_value_toks` to ⟨*value*⟩, or ⟨*default*⟩ if ⟨*value*⟩ was not given and if ⟨*default*⟩ is available.

`\keys_value_requirement:n` `\keys_value_requirement:nn` ⟨*requirement*⟩

Sets ⟨*key*⟩ to have ⟨*requirement*⟩ concerning ⟨*value*⟩.

`\keys_variable_set:NN` `\keys_variable_set:NN` ⟨*expansion*⟩ ⟨*var*⟩

Sets ⟨*key*⟩ to assign ⟨*value*⟩ to ⟨*variable*⟩

`\keys_variable_get_scope:N` ⋆
`\keys_variable_get_type:N` ⋆ `\keys_variable_get_scope:N` ⟨*var*⟩

Returns the scope (g or blank) or the type of ⟨*var*⟩.

1.7 Variables and constants

`\c_keys_properties_root_tl`
`\c_keys_root_tl` The root paths for keys and properties.

`\c_keys_value_forbidden_tl`
`\c_keys_value_required_tl` Marker text containers.

`\l_keys_module_tl`
`\l_keys_path_tl` Various key paths need to be stored.

`\l_keys_no_value_bool` A marker for "no value" as key input.

`\l_keys_value_toks` Holds the currently supplied value.

The usual preliminaries.

```
1 ⟨*package⟩
2 \ProvidesExplPackage
3   {\filename}{\filedate}{\fileversion}{\filedescription}
```

1.7.1 Variables and constants

`\c_keys_root_tl`
`\c_keys_properties_root_tl` Where the keys are really stored.

```
4 \tl_new:Nn \c_keys_root_tl { keys~>~ }
5 \tl_new:Nn \c_keys_properties_root_tl { keys_properties }
```

`\c_keys_value_forbidden_tl`
`\c_keys_value_required_tl` Two marker token lists.

```
6 \tl_new:Nn \c_keys_value_forbidden_tl { forbidden }
7 \tl_new:Nn \c_keys_value_required_tl { required }
```

`\l_keys_choice_int`
`\l_keys_choice_tl` Used for the multiple choice system.

```
8 \int_new:N \l_keys_choice_int
9 \tl_new:N \l_keys_choice_tl
```

`\l_keys_key_tl`
`\l_keys_path_tl` Storage for the current key name and the path of the key (key name plus module name).

```
10 \tl_new:N \l_keys_key_tl
11 \tl_new:N \l_keys_path_tl
```

`\l_keys_module_tl` The module for an entire set of keys.

```
\tl_new:N \l_keys_module_tl
```

`\l_keys_no_value_bool` To indicate that no value has been given.

```
\bool_new:N \l_keys_no_value_bool
```

`\l_keys_value_toks` A token register for the given value.

```
\toks_new:N \l_keys_value_toks
```

1.7.2 Internal functions

`\keys_bool_set:N` `\keys_bool_set_inverse:N` `\keys_bool_set_aux:N` Boolean keys are really just choices, but all done by hand.

```
\cs_new_nopar:Nn \keys_bool_set:N {
  \keys_cmd_set:nNx { \l_keys_path_tl / true } 1 {
    \exp_not:c { bool_ \keys_variable_get_scope:N #1 set_true:N }
      \exp_not:N #1
  }
  \keys_cmd_set:nNx { \l_keys_path_tl / false } 1 {
    \exp_not:N \use:c
      { bool_ \keys_variable_get_scope:N #1 set_false:N }
      \exp_not:N #1
  }
  \keys_bool_set_aux:N #1
}
\cs_new_nopar:Nn \keys_bool_set_inverse:N {
  \keys_cmd_set:nNx { \l_keys_path_tl / true } 1 {
    \exp_not:c { bool_ \keys_variable_get_scope:N #1 set_false:N }
      \exp_not:N #1
  }
  \keys_cmd_set:nNx { \l_keys_path_tl / false } 1 {
    \exp_not:c { bool_ \keys_variable_get_scope:N #1 set_true:N }
      \exp_not:N #1
  }
  \keys_bool_set_aux:N #1
}
\cs_new_nopar:Nn \keys_bool_set_aux:N {
  \keys_choice_make:
  \cs_if_exist:NF #1 {
    \bool_new:N #1
  }
  \keys_default_set:n { true }
}
```

`\keys_choice_find:n` Executing a choice has two parts. First, try the choice given, then if that fails call the unknown key. That will exist, as it is created when a choice is first made. So there is no need for any escape code.

```
\cs_new_nopar:Nn \keys_choice_find:n {
  \keys_execute_aux:nn { \l_keys_path_tl / #1 } {
```

```
    \keys_execute_aux:nn { \l_keys_path_tl / unknown } { }
  }
}
```

`\keys_choice_make:` To make a choice from a key, two steps: set the code, and set the unknown key.

```
\cs_new_nopar:Nn \keys_choice_make: {
  \keys_cmd_set:nNn { \l_keys_path_tl } 1 {
    \keys_choice_find:n {##1}
  }
  \keys_cmd_set:nNn { \l_keys_path_tl / unknown } 1 {
    \msg_error:nnxx { keys } { choice~unknown }
      { \l_keys_path_tl } {##1}
  }
}
```

`\keys_choices_generate:nx` `\keys_choices_generate_aux:n` Creating multiple-choices means setting up the "indicator" code, then applying whatever the user wanted.

```
\cs_new:Nn \keys_choices_generate:nx {
  \keys_choice_make:
  \int_zero:N \l_keys_choice_int
  \cs_set_nopar:Nn \keys_choices_generate_aux:n {
    \int_incr:N \l_keys_choice_int
    \keys_cmd_set:nNx { \l_keys_path_tl / ##1 } 1 {
      \exp_not:n { \tl_set:Nn \l_keys_choice_tl } {##1}
      \exp_not:n { \int_set:Nn \l_keys_choice_int }
        { \int_use:N \l_keys_choice_int }
      #2
    }
  }
  \clist_map_function:nN {#1} \keys_choices_generate_aux:n
}
\cs_new_nopar:Nn \keys_choices_generate_aux:n { }
```

`\keys_cmd_set:nNn` `\keys_cmd_set:nNx` `\keys_cmd_set_aux:nN` Creating a new command means setting properties and then creating a function with the correct number of arguments.

```
\cs_new:Nn \keys_cmd_set:nNn {
  \keys_cmd_set_aux:nN {#1} #2
  \cs_generate_from_arg_count:cNnn { \c_keys_root_tl #1 .cmd:w }
    \cs_set:Npn #2 {#3}
}
\cs_new:Nn \keys_cmd_set:nNx {
  \keys_cmd_set_aux:nN {#1} #2
  \cs_generate_from_arg_count:cNnn { \c_keys_root_tl #1 .cmd:w }
    \cs_set:Npx #2 {#3}
}
\cs_new_nopar:Nn \keys_cmd_set_aux:nN {
  \keys_property_undefine:n { #1 .default_tl }
  \num_set:cn { \c_keys_root_tl #1 .args_num } {#2}
  \tl_set:cn  { \c_keys_root_tl #1 .req_tl } { }
}
```

\keys_default_set:n
\keys_default_set:V

Setting a default value is easy.

```
\cs_new:Nn \keys_default_set:n {
  \tl_set:cn { \c_keys_root_tl \l_keys_path_tl .default_tl } {#1}
}
\cs_generate_variant:Nn \keys_default_set:n { V }
```

\keys_define:nn

The main key-defining function mainly sets up things for l3keyval to use.

```
\cs_new:Nn \keys_define:nn {
  \tl_set:Nn \l_keys_module_tl {#1}
  \cs_set_eq:NN \KV_key_no_value_elt:n \keys_define_elt:n
  \cs_set_eq:NN \KV_key_value_elt:nn \keys_define_elt:nn
  \KV_parse_no_space_removal_no_sanitize:n {#2}
}
```

\keys_define_elt:n
\keys_define_elt:nn

The element processors for defining keys.

```
\cs_new:Nn \keys_define_elt:n {
  \bool_set_true:N \l_keys_no_value_bool
  \keys_define_elt_aux:nn {#1} { }
}
\cs_new:Nn \keys_define_elt:nn {
  \bool_set_false:N \l_keys_no_value_bool
  \keys_define_elt_aux:nn {#1} {#2}
}
```

\keys_define_elt_aux:nn

The auxiliary function does most of the work.

```
\cs_new:Nn \keys_define_elt_aux:nn {
  \keys_property_find:n {#1}
  \cs_set_eq:Nc \keys_tmp:w
    { \c_keys_properties_root_tl \l_keys_key_tl }
  \cs_if_exist:NTF \keys_tmp:w {
    \keys_define_key:n {#2}
  }{
    \msg_error:nnx { keys } { property~unknown }
      { \l_keys_key_tl }
  }
}
```

\keys_define_key:n

Defining a new key means finding the code for the appropriate property then running it. As properties have signatures, a check can be made for required values without needing anything set explicitly.

```
\cs_new:Nn \keys_define_key:n {
  \bool_if:NTF \l_keys_no_value_bool {
    \intexpr_compare:nTF {
      \exp_args:Nc \cs_get_arg_count_from_signature:N
        { \l_keys_key_tl }  = \c_zero
    } {
      \keys_tmp:w
    }{
```

```
      \msg_error:nnx { keys } { property~value~required }
        { \l_keys_key_tl }
    }
  }{
    \intexpr_compare:nTF {
      \exp_args:Nc \cs_get_arg_count_from_signature:N
        { \l_keys_key_tl }  = \c_one
    } {
      \keys_tmp:w {#1}
    }{
      \keys_tmp:w #1
    }
  }
}
```

`\keys_execute:`
`\keys_execute_unknown:`
`\keys_execute_aux:nn`

Actually executing a key is done in two parts. First, look for the key itself, then look for the `unknown` key with the same path. If both of these fail, complain!

```
\cs_new_nopar:Nn \keys_execute: {
  \keys_execute_aux:nn { \l_keys_path_tl } {
    \keys_execute_unknown:
  }
}
\cs_new_nopar:Nn \keys_execute_unknown: {
  \keys_execute_aux:nn { \l_keys_module_tl / unknown } {
    \msg_error:nnx { keys } { key~unknown } { \l_keys_path_tl }
  }
}
```

If there is only one argument required, it is wrapped in braces so that everything is passed through properly. On the other hand, if more than one is needed it is down to the user to have put things in correctly!

```
\cs_new_nopar:Nn \keys_execute_aux:nn {
  \cs_set_eq:Nc \keys_tmp:w { \c_keys_root_tl #1 .cmd:w }
  \cs_if_exist:NTF \keys_tmp:w {
    \intexpr_compare:nTF {
      \num_use:c { \c_keys_root_tl #1 .args_num } = \c_one
    } {
      \exp_args:NV \keys_tmp:w \l_keys_value_toks
    }{
      \exp_after:wN \keys_tmp:w \toks_use:N \l_keys_value_toks
    }
  }{
    #2
  }
}
```

`keys_if_value_requirement:nTF`

To test if a value is required or forbidden. Only one version is needed, so done by hand.

```
\cs_new_nopar:Npn \keys_if_value_requirement:nTF #1 {
  \tl_if_eq:ccTF { c_keys_value_ #1 _tl } {
    \c_keys_root_tl \l_keys_path_tl .req_tl
  }
}
```

\keys_property_find:n
\keys_property_find_aux:w

Searching for a property means finding the last "." in the input, and storing the text before and after it.

```
\cs_new_nopar:Nn \keys_property_find:n {
  \tl_set:Nx \l_keys_path_tl { \l_keys_module_tl / }
  \tl_if_in:nnTF {#1} {.} {
    \keys_property_find_aux:n {#1}
  }{
    \msg_error:nnx { keys } { no~property } { #1 }
  }
}
\cs_new_nopar:Nn \keys_property_find_aux:n {
  \keys_property_find_aux:w #1 \q_stop
}
\cs_new_nopar:Npn \keys_property_find_aux:w #1 . #2 \q_stop {
  \tl_if_in:nnTF {#2} {.} {
    \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl . #1 }
    \keys_property_find_aux:w #2 \q_stop
  }{
    \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl \tl_to_str:n {#1} }
    \tl_set:Nn \l_keys_key_tl { . #2 }
  }
}
```

\keys_property_new:nn

Creating a new propery is simply a case of making the correctly-named function.

```
\cs_new_nopar:Nn \keys_property_new:nn {
  \cs_new:cn { \c_keys_properties_root_tl #1 } {#2}
}
```

\keys_property_undefine:n

Removing a property means undefining it.

```
\cs_new_nopar:Nn \keys_property_undefine:n {
  \cs_set_eq:cN { \c_keys_root_tl #1 } \c_undefined
}
```

\keys_set:nn

The main setting function just does the set up to get l3keyval to do the hard work.

```
\cs_new:Nn \keys_set:nn {
  \tl_set:Nn \l_keys_module_tl {#1}
  \cs_set_eq:NN \KV_key_no_value_elt:n \keys_set_elt:n
  \cs_set_eq:NN \KV_key_value_elt:nn \keys_set_elt:nn
  \KV_parse_space_removal_sanitize:n {#2}
}
```

\keys_set_elt:n
\keys_set_elt:nn

The two element processors are almost identical, and pass the data through to the underlying auxiliary, which does the work.

```
\cs_new:Nn \keys_set_elt:n {
  \bool_set_true:N \l_keys_no_value_bool
  \keys_set_elt_aux:nn {#1} { }
}
\cs_new:Nn \keys_set_elt:nn {
```

```
  \bool_set_false:N \l_keys_no_value_bool
  \keys_set_elt_aux:nn {#1} {#2}
}
```

`\keys_set_elt_aux:nn`
`\keys_set_elt_aux:`

First, set the current path and add a default if needed. There are then checks to see if the a value is required or forbidden. If everything passes, move on to execute the code.

```
\cs_new:Nn \keys_set_elt_aux:nn {
  \tl_set:Nx \l_keys_key_tl { \tl_to_str:n {#1} }
  \tl_set:Nx \l_keys_path_tl { \l_keys_module_tl / \l_keys_key_tl }
  \keys_value_or_default:n {#2}
  \keys_if_value_requirement:nTF { required } {
    \bool_if:NTF \l_keys_no_value_bool {
      \msg_error:nnx { keys }  { key~value~required }
        { \l_keys_path_tl }
    }{
      \keys_set_elt_aux:
    }
  }{
    \keys_set_elt_aux:
  }
}
\cs_new_nopar:Nn \keys_set_elt_aux: {
  \keys_if_value_requirement:nTF { forbidden } {
    \bool_if:NTF \l_keys_no_value_bool {
      \keys_execute:
    }{
      \msg_error:nnxx { keys} { key~value~forbidden }
        { \l_keys_path_tl }
        { \toks_use:N \l_keys_value_toks }
    }
  }{
    \keys_execute:
  }
}
```

`\keys_show:nn` Showing a key is just a question of using the correct name.

```
\cs_new_nopar:Nn \keys_show:nn {
  \cs_show:c { \c_keys_root_tl #1 / \tl_to_str:n {#2} .cmd:w }
}
```

`\keys_tmp:w` This scratch function is used to actually execute keys.

```
\cs_new:Npn \keys_tmp:w {}
```

`\keys_value_or_default:n` If a value is given, return it as `#1`, otherwise send a default if available.

```
\cs_new:Nn \keys_value_or_default:n {
  \toks_set:Nn \l_keys_value_toks  {#1}
  \bool_if:NT \l_keys_no_value_bool {
    \cs_if_exist:cT { \c_keys_root_tl \l_keys_path_tl .default_tl } {
      \toks_set:Nv \l_keys_value_toks  {
```

```
        \c_keys_root_tl \l_keys_path_tl .default_tl
      }
    }
  }
}
```

`\keys_value_requirement:n` Values can be required or forbidden by having the appropriate marker set.

```
\cs_new_nopar:Nn \keys_value_requirement:n {
  \tl_set_eq:cc { \c_keys_root_tl \l_keys_path_tl .req_tl }
    { c_keys_value_ #1 _tl }
}
```

`\keys_variable_get_scope:N` `\keys_variable_get_scope_aux:N` `\keys_variable_get_type:N` `\keys_variable_get_type:w` Expandable functions to find the type of a variable, and to return `g` if the variable is global.

```
\cs_new_nopar:Nn \keys_variable_get_scope:N {
  \tl_if_eq:xxT { \token_to_str:N g }
    { \keys_variable_get_scope_aux:N  #1 }
    { g }
}
\cs_new_nopar:Nn \keys_variable_get_scope_aux:N {
  \exp_last_unbraced:NNo \use_i:nn \use_i_delimit_by_q_stop:nw
    \token_to_str:N #1 \q_stop
}
\group_begin:
  \char_set_lccode:nn {`\&} {`\_}
  \char_make_other:N \&
\tl_to_lowercase:n {
  \group_end:
  \cs_new_nopar:Nn \keys_variable_get_type:N {
    \exp_after:wN \keys_variable_get_type_aux:w
      \token_to_str:N #1 & \q_nil \q_stop
  }
  \cs_new_nopar:Npn \keys_variable_get_type_aux:w #1 & #2 \q_stop {
    \quark_if_nil:nTF {#2} {
      #1
    }{
      \keys_variable_get_type_aux:w #2 \q_stop
    }
  }
}
```

`\keys_variable_set:NN` To set a variable, there is first a check so that it must exist. The setting function is then created by recovering the type and scope from the variable name.

```
\cs_new_nopar:Nn \keys_variable_set:NN {
  \cs_if_exist:NF #2 {
    \use:c { \keys_variable_get_type:N #2 _new:N } #2
  }
  \keys_cmd_set:nNx { \l_keys_path_tl } 1 {
    \exp_not:c {
      \keys_variable_get_type:N #2 _
```

```
      \keys_variable_get_scope:N #2 set:N #1
    } \exp_not:N #2 {##1}
  }
}
```

`.choice:` Making a choice is handled internally, as it is also needed by `.generate_choices:nn`.

```
\keys_property_new:nn { .choice: } {
  \keys_choice_make:
}
```

`.code:n` `.code:x` `.code:Nn` `.code:Nx` Creating code is simply a case of passing through to the underlying `set` function.

```
\keys_property_new:nn { .code:n } {
  \keys_cmd_set:nNn { \l_keys_path_tl } 1 {#1}
}
\keys_property_new:nn { .code:Nn } {
  \keys_cmd_set:nNn { \l_keys_path_tl } #1 {#2}
}
\keys_property_new:nn { .code:x } {
  \keys_cmd_set:nNx { \l_keys_path_tl } 1 {#1}
}
\keys_property_new:nn { .code:Nx } {
  \keys_cmd_set:nNx { \l_keys_path_tl } #1 {#2}
}
```

`.default:n` `.default:V` Expansion is left to the internal functions.

```
\keys_property_new:nn { .default:n } {
  \keys_default_set:n  {#1}
}
\keys_property_new:nn { .default:V } {
  \keys_default_set:V #1
}
```

`.generate_choices:nn` `.generate_choices:nx` Making choices is expansion-dependent.

```
\keys_property_new:nn { .generate_choices:nn } {
  \keys_choices_generate:nx {#1} { \exp_not:n {#2} }
}
\keys_property_new:nn { .generate_choices:nx } {
  \keys_choices_generate:nx {#1} {#2}
}
```

1.7.3 Properties

`.set:N` `.set_x:N` Setting a variable is very easy: just pass the data along.

```
\keys_property_new:nn { .set:N } {
  \keys_variable_set:NN n #1
}
\keys_property_new:nn { .set_x:N } {
  \keys_variable_set:NN x #1
}
```

`.set_bool:N`
`.set_bool_inverse:N`

One function for each of these, but this keeps the key functions themselves short.

```
\keys_property_new:nn { .set_bool:N } {
  \keys_bool_set:N #1
}
\keys_property_new:nn { .set_bool_inverse:N } {
  \keys_bool_set_inverse:N #1
}
```

`.value_forbidden:`
`.value_required:`

These are very similar, so both call the same function.

```
\keys_property_new:nn { .value_forbidden: } {
  \keys_value_requirement:n { forbidden }
}
\keys_property_new:nn { .value_required: } {
  \keys_value_requirement:n { required }
}
```

1.7.4 Messages

For when the package needs to complain.

```
\msg_new:nnn { keys } { choice~unknown } {%
  Choice ‘#2’ unknown for key ‘#1’:\\%
  the key is being ignored.%
}
\msg_new:nnn { keys } { key~unknown } {%
  The key ‘#1’ is unknown and is being ignored.%
}
\msg_new:nnn { keys } { key~value~forbidden }{%
  The key ‘#1’ cannot taken a value:\\%
  the given input ‘#2’ is being ignored.%
}
\msg_new:nnn { keys } { key~value~required } {%
  The key ‘#1’ requires a value\\%
  and is being ignored.%
}
\msg_new:nnn { keys } { no~property } {%
  No property given in definition of key ‘#1’.%
}
\msg_new:nnn { keys } { property~unknown } {%
  The key property ‘#1’ is unknown.%
}
\msg_new:nnn { keys } { property~value~required } {%
  The property ‘#1’ requires a value\\%
  and is being ignored.%
}
```

⟨/package⟩