

The `keys3` package*

Key management for L^AT_EX₃

Joseph Wright[†]

2009/03/12

1 Key management

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

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

or

```
\PackageMacro[
  key      = value,
  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. However, the underlying model is rather inflexible.

As an alternative, `pgfkeys` from the `pgf/Tikz` bundle uses a “file-like” model for keys. In this model, each key has one or more functions attached to define its action. In `pgfkeys` terminology, these are key *handlers*. Keys can be created and used using a unified interface:

*This file has version number 75, last revised 2009/03/12.

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

```

\pgfkeys{
  /path/key/.handler = code,
  /path/key           = value
}

```

The `keys3` package is aimed at creating a high-level programmers interface for key–value controls in $\text{\LaTeX}3$. Key creation and control follows the `pgfkeys` closely, although changes have been made to adhere to the new coding ideas of $\text{\LaTeX}3$, and new functions have been added. The `keys3` package is *not* a simple translation of `pgfkeys` to the new syntax. In package internals have been written from the ground up, to better enforce variable typing and to act in a know manner under failing circumstances.

In the `keys3` model, each key belongs to a module, which may have one or more subdivisions. Each key then has one or more properties which define how the key acts. The idea of a single function for all key control is retained:

```

\keys_manage:n{
  /module/key/.property      = code,
  /module/sub/key/.property = more code,
  /module/key                = value
}

```

The combination of all $\langle module \rangle$ parts up to the $\langle key \rangle$ is referred to as the *path* of the key.

1.1 Creating, retrieving and setting keys

The main interface for key management is the the `\keys_manage:n` macro. This can be used to create, retrieve and set keys, and is therefore the preferred interface for the package.

<pre> \keys_manage:n \keys_manage_quick:n \keys_manage_internal:n </pre>	<code>\keys_manage:n</code> $\langle keyval list \rangle$
--	---

The main key management function, which parses over the $\langle keyval list \rangle$ and processes each key. The standard version removes leading and trailing spaces and checks catcodes for ungrouped “,” and “=”. The `_quick` version omits these tests for speed, and is therefore more suited to creating keys inside code blocks. The `internal` variant is used when recycling keys inside `keys3`.

<pre> \keys_manage:nn \keys_manage_quick:nn </pre>	<code>\keys_manage:nn</code> $\langle path \rangle$ $\langle keyval list \rangle$
--	---

Versions of the above which include a $\langle path \rangle$ as an argument. These are faster versions of the `:n` macros when setting lots of keys separately and with known paths.

Setting up and altering keys is carried out using one or more properties (in `pgfkeys`, these are called handlers). In all cases, $\langle key \rangle$ may be a full-qualified key with a path, or a partial key to which the default path will be added. Notice that when a single argument is required, the braces around the argument may be omitted without any error arising.

1.1.1 Storing values

A common use of key-value input is to store the values given in a variable for later use. `keys3` provides pre-defined properties for storing in `int`, `skip`, `tlp` and `toks` variables. The method is similar in all cases:

```
\keys_manage:n{
  /module/key~one/.int_set:N = \l_mod_data_int,
  /module/key~two/.skip_set:N = \l_mod_data_skip,
  /module/key~three/.tlp_set:N = \l_mod_data_tlp,
  /module/key~four/.toks_set:N = \l_mod_data_toks
}
```

Global variants for all of these are.

When using keys to store a value, it is often convenient to give only the unique part of the variable name. This is particularly true when creating a large number of related storage areas. To achieve this, `keys3` requires that the module prefix to be used is defined first. The special key `/keys/current_module:n` can be set with the name of the current module.

```
\keys_manage:n{
  /keys/current_module:n = module,
  /module/key/.tlp_set:n = my_data,
}
```

This will use a variable called `\l_module_my_data_tlp` to store the input. Thus the preceding code achieves the same effect as

```
\keys_manage:n{
  /module/key/.tlp_set:N = \l_module_my_data_tlp
}
```

The same method applies to `int`, `skip` and `toks` variables. The appropriate prefix (`\l_` or `\g_`) and suffix (`_int`, `_skip`, `_tlp` or `_toks`) is always added.

1.1.2 Multiple choice keys

Multiple choice keys are created in `keys3` using the `.expects_choice:` property. Each choice is then a sub-key of the choice key.

```
\keys_manage:n{
  /module/key/.expects_choice:,
  /module/key/choice~a/.code:n = Some code,
  /module/key/choice~b/.code:n = Some other code,
}
```

In this way, choices which execute arbitrary code can be created. Notice that the `<code>` should *not* include a parameter (`#1`).

Often it is desirable to create a family of similar choices, which only require either the text of the choice, or the position of the choice in a list, to be used. To create this type of simple choice, `keys3` provides the `.create_choices:nn` property. This applies the same code to a list of choice text. Inside the code, the name of the choice given is available as `\l_keys_current_choice_tlp`. The position of the choice in the lists is also available, as `\l_keys_current_choice_int`.

```
\keys_manage:n {
  /module/key/.create_choices:nn = {choice~a, choice~b, choice~c} {
    You-gave-choice~'\l_keys_current_choice_tlp',~which-is-in~
    position~\l_keys_current_choice_int~in~the~list.
  }
}
```

1.2 Properties

```
.bool_set:N
.bool_gset:N <key>/bool_set:N = <bool>
```

Defines `<key>` to set `<bool>` to `<value>` (which must be either `true` or `false`).

```
.bool_set_inverse:n
.bool_gset_inverse:n <key>/bool_set_inverse:n = <name>
```

Defines `<key>` to set switch with unique `<name>` to `<value>` (which must be either `true` or `false`), with reversed logic. The switch name will be constructed using the current `<module>` (if any), prefixed by `\l_` and ending with `_int`.

```
.bool_set_inverse:N
.bool_gset_inverse:N <key>/bool_set_inverse:N = <bool>
```

Defines $\langle key \rangle$ to set $\langle bool \rangle$ to $\langle value \rangle$ (which must be either `true` or `false`), with reversed logic.

```
.bool_set:n  
.bool_gset:n  $\langle key \rangle$ /.bool_set:n =  $\langle name \rangle$ 
```

Defines $\langle key \rangle$ to set switch with unique $\langle name \rangle$ to $\langle value \rangle$ (which must be either `true` or `false`). The switch name will be constructed using the current $\langle module \rangle$ (if any), prefixed by `\1_` and ending with `_int`.

```
.cd:  $\langle key \rangle$ /.cd:
```

Changes path to that given by $\langle key \rangle$.

```
.code:n  
.code:x  $\langle key \rangle$ /.code:n =  $\langle code \rangle$ 
```

Stores the $\langle code \rangle$ for execution when $\langle key \rangle$ is called. The $\langle code \rangle$ can include one parameter (`#1`).

```
.code:Nn  
.code:Nx  $\langle key \rangle$ /.code:Nn =  $\langle number \rangle$   $\langle code \rangle$ 
```

Stores the $\langle code \rangle$ for execution when $\langle key \rangle$ is called. The $\langle code \rangle$ can include $\langle number \rangle$ parameters, which can be in the range 0–9.

```
.create_choices:nn  
.create_choices:nx  $\langle key \rangle$ /.create_choices:nn =  $\langle list \rangle$   $\langle code \rangle$ 
```

Creates a sub-key of $\langle key \rangle$ for each $\langle choice \rangle$ in the comma-separated $\langle list \rangle$. Each $\langle choice \rangle$ will have $\langle code \rangle$ associated with it. The current choice text is available as `\1_keys_current_choice_tlp`, and its position in the $\langle list \rangle$ as `\1_keys_current_choice_int`.

```
.default:n  $\langle key \rangle$ /.default:n =  $\langle content \rangle$ 
```

Creates a default value for $\langle key \rangle$, which is used if no value is given. The $\langle content \rangle$ is stored as a tlp, and so must be compatible with this variable type.

```
.expects_choice:  $\langle key \rangle$ /.expects_choice:
```

Indicates that the $\langle value \rangle$ given for $\langle key \rangle$ should be a sub-key of $\langle key \rangle$. In this way, $\langle key \rangle$ accepts one of a limited range of choices.

```
.int_set:N  
.int_gset:N  $\langle key \rangle$ /.int_set:N =  $\langle int \rangle$ 
```

Defines $\langle key \rangle$ to store $\langle value \rangle$ in the $\langle int \rangle$ named.

```
.int_set:n
.int_gset:n <key>/int_set:n = <name>
```

Defines $\langle key \rangle$ to store $\langle value \rangle$ in a int with unique name $\langle name \rangle$. The int name will be constructed using the current $\langle module \rangle$ (if any), prefixed by $\backslash 1_$ and ending with $_int$.

```
.retry:n <key>/retry:n = <data>
```

Executes $\langle key \rangle$ if it exists and if the previous $\.try:n$ failed. The $\langle data \rangle$ is passed to $\langle key \rangle$, if successful.

```
.show_code: <key>/show_code:
.show_key: <key>/show_key:
```

Shows the function for the current key or the code associated with the current key.

```
.skip_set:N
.skip_gset:N <key>/skip_set:N = <skip>
```

Defines $\langle key \rangle$ to store $\langle value \rangle$ in the $\langle skip \rangle$ named.

```
.skip_set:n
.skip_gset:n <key>/skip_set:n = <name>
```

Defines $\langle key \rangle$ to store $\langle value \rangle$ in a skip with unique name $\langle name \rangle$. The skip name will be constructed using the current $\langle module \rangle$ (if any), prefixed by $\backslash 1_$ and ending with $_skip$.

```
.tlp_set:N
.tlp_set_x:N
.tlp_gset:N
.tlp_gset_x:N <key>/tlp_set:N = <tlp>
```

Defines $\langle key \rangle$ to store $\langle value \rangle$ in the $\langle tlp \rangle$ named. The x versions use $\backslash tlp_g)set:Nx$ for this process, the standard version $\backslash tlp_g)set:Nn$.

```
.tlp_set:n
.tlp_set_x:n
.tlp_gset:n
.tlp_gset_x:n <key>/tlp_set:n = <name>
```

Defines $\langle key \rangle$ to store $\langle value \rangle$ in a tlp with unique name $\langle name \rangle$. The tlp name will be constructed using the current $\langle module \rangle$ (if any), prefixed by $\backslash 1_$ and ending with $_tlp$.

```
.toks_set:N
.toks_gset:N <key>/toks_set:N = <toks>
```

Defines $\langle key \rangle$ to store $\langle value \rangle$ in the $\langle toks \rangle$ named.

```
.toks_set:n  
.toks_gset:n
```

```
<key>/toks_set:n = <name>
```

Defines *<key>* to store *<value>* in a toks with unique name *<name>*. The rest of the toks name will be constructed using the current *<module>* (if any), prefixes by `\l_` and ending with `_toks`.

```
.try:n
```

```
<key>/try:n = <data>
```

Executes *<key>* if defined, and does nothing otherwise. The *<data>* is passed to *<key>*, if successful.

```
.use_keys:n
```

```
.use_keys:x
```

```
<key>/use_keys:n = <keyval list>
```

Calling *<key>* applies the *<keyval list>* as a block. Thus one *<key>* can make many related settings. As usual, `#1` is available in the *<keyval list>* as *<key>* will take one argument.

```
.use_keys:Nn
```

```
.use_keys:Nx
```

```
<key>/use_keys:Nn = <number> <keyval list>
```

Calling *<key>* applies the *<keyval list>* as a block. Thus one *<key>* can make many related settings. The *<key>* will take *<number>* of arguments, which can be used inside the *<keyval list>*.

```
.value_forbidden:
```

```
.value_required:
```

```
<key>/value_forbidden:
```

Flags for forbidding and requiring a *<value>* for *<key>*.

1.3 Variables and constants

```
\c_keys_0_empty_tlp
```

```
\c_keys_1_empty_tlp
```

```
\c_keys_2_empty_tlp
```

```
\c_keys_3_empty_tlp
```

```
\c_keys_4_empty_tlp
```

```
\c_keys_5_empty_tlp
```

```
\c_keys_6_empty_tlp
```

```
\c_keys_7_empty_tlp
```

```
\c_keys_8_empty_tlp
```

```
\c_keys_9_empty_tlp
```

A set of tpls containing empty groups.

```
\c_keys_cs_prefix_tlp
```

The prefix added to the fully-qualified key when saving them.

`\c_keys_errors_path_tlp`
`\c_keys_properties_path_tlp`
`\c_keys_utilities_path_tlp` Paths for properties used by l3keys itself.

`\c_keys_root_tlp` The root path for keys.

`\l_keys_choice_code_tlp` The code to execute for each multiple choice when created *en masse*.

`\l_keys_current_choice_tlp`
`\l_keys_current_choice_int` Information on multiple choices.

`\l_keys_current_key_full_tlp`
`\l_keys_current_key_name_tlp` The current key name is stored both with and without a path.

`\l_keys_current_module_tlp` Current module name used when creating csnames.

`\l_keys_current_path_tlp`
`\l_keys_default_path_tlp`
`\l_keys_choice_path_tlp` Various key paths need to be stored.

`\l_keys_current_value_toks` The value given for the current key, stored as a token register.

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

`\l_keys_success_bool` A marker used when trying keys without raising errors.

`\l_keys_tmpa_tlp` A scratch variable.

`\l_keys_err_unknown_key_tlp`
`\l_keys_err_value_ignored_tlp`
`\l_keys_err_value_required_tlp`
`\l_keys_err_def_x_args_tlp`
`\l_keys_err_boolean_expected_tlp`
`\l_keys_err_not_boolean_tlp`
`\l_keys_err_unknown_choice_tlp` Identification tpls for error messages.

1.4 Internal functions

Notice that everything should be done using the keys system. Only `\keys_manage:n` and so on are intended for external use. In all cases, $\langle key \rangle$ is a fully-qualified key name. Functions created will be prefixed with `\c_keys_cs_prefix_tlp`.

<code>\keys_bool_new:n</code>	<code>\keys_bool_new:n</code> $\langle key \rangle$
-------------------------------	---

Creates a switch $\langle key \rangle$, which will be set to `true`.

<code>\keys_bool_set:nN</code>	<code>\keys_bool_set:nN</code> $\langle function \rangle$ $\langle boolean \rangle$
<code>\keys_bool_set:nnn</code>	<code>\keys_bool_set:nnn</code> $\langle function \rangle$ $\langle prefix \rangle$ $\langle name \rangle$

Uses $\langle function \rangle$ (either `set` or `gset`) to use $\langle value \rangle$ in setting $\langle boolean \rangle$. If a $\langle name \rangle$ is given, it is used to construct a csname for the switch including the current $\langle module \rangle$. The $\langle prefix \rangle$ should be `l_` or `g_`.

<code>\keys_bool_set_inverse:n</code>	<code>\keys_bool_set:n</code> $\langle switch \rangle$
<code>\keys_bool_set_inverse:nN</code>	<code>\keys_bool_set:nN</code> $\langle function \rangle$ $\langle boolean \rangle$
<code>\keys_bool_set_inverse:nnn</code>	<code>\keys_bool_set:nnn</code> $\langle function \rangle$ $\langle prefix \rangle$ $\langle name \rangle$

Uses $\langle function \rangle$ (either `set` or `gset`) to use $\langle value \rangle$ in setting $\langle boolean \rangle$. If a $\langle name \rangle$ is given, it is used to construct a csname for the switch including the current $\langle module \rangle$. The $\langle prefix \rangle$ should be `l_` or `g_`. The logic of the setting is reversed compared to the `\keys_bool_set:` functions.

<code>\keys_clear_properties:n</code>	<code>\keys_clear_properties:n</code> $\langle key \rangle$
---------------------------------------	---

Clears the internal properties of $\langle key \rangle$.

<code>\keys_choice_create:n</code>	<code>\keys_choice_create:n</code> $\langle choice \rangle$
------------------------------------	---

Creates $\langle choice \rangle$ as a sub-key of $\langle key \rangle$.

<code>\keys_choices_create:Nnn</code>	<code>\keys_choices_create:Nnn</code> $\langle expansion \rangle$ $\langle list \rangle$ $\langle code \rangle$
---------------------------------------	---

Takes a comma separated $\langle list \rangle$ and makes a $\langle choice \rangle$ as a sub-key of the current $\langle key \rangle$ for each. The $\langle choice \rangle$ will execute $\langle code \rangle$, which is processed according to $\langle expansion \rangle$ (either `n` or `x`). The $\langle code \rangle$ has access to `\l_keys_current_choice_tlp` and `\l_keys_current_choice_int`, which indicate the choice given.

<code>\keys_default_add:</code>	<code>\keys_default_add:</code>
---------------------------------	---------------------------------

If no value was given for the current key, and a default value is available, copies the default into `\l_keys_current_value_toks`.

<code>\keys_err_new:nNnnn</code>	<code>\keys_err_new:nNnnn</code> $\langle name \rangle$ $\langle args \rangle$ $\langle short \rangle$ $\langle long \rangle$ $\langle code \rangle$
----------------------------------	--

Creates error with $\langle name \rangle$ and taking $\langle args \rangle$ arguments, with $\langle short \rangle$ description followed by $\langle long \rangle$, and recovery $\langle code \rangle$.

<code>\keys_err_use:nw</code>	
<code>\keys_err_use:n</code>	<code>\keys_err_use:n</code> $\langle name \rangle$
<code>\keys_err_use:nn</code>	<code>\keys_err_use:nn</code> $\langle name \rangle$ $\langle content \rangle$

Issues error $\langle name \rangle$, with or without $\langle content \rangle$. The `w` variant may take one or two arguments.

<code>\keys_find_code_full:</code>	<code>\keys_find_code_full:</code>
<code>\keys_find_code_name:</code>	<code>\keys_find_code_name:</code>

Find code to execute for the current fully qualified key (`full`) or key name only (`name`).

<code>\keys_if_cmd_really_exist:nTF</code>	<code>\keys_if_cmd_really_exist:nTF</code> $\langle key \rangle$ $\langle true code \rangle$ $\langle false code \rangle$
--	--

Checks for the existence of a `._cmd:w` function for $\langle key \rangle$.

<code>\keys_if_really_exist:nTF</code>	<code>\keys_if_really_exist:nTF</code> $\langle key \rangle$ $\langle true code \rangle$ $\langle false code \rangle$
--	--

Checks for the existence of a $\langle key \rangle$.

<code>\keys_if_value:nTF</code>	<code>\keys_if_value:nTF</code> $\langle property \rangle$ $\langle true code \rangle$ $\langle false code \rangle$
---------------------------------	---

Checks for the $\langle property \rangle$ switch of a $\langle key \rangle$ (typically `required` or `forbidden`).

<code>\keys_no_value_elt:n</code>	<code>\keys_no_value_elt:n</code> $\langle elt \rangle$
<code>\keys_value_elt:nn</code>	<code>\keys_value_elt:nn</code> $\langle elt \rangle$ $\langle value \rangle$

Functions used by `l3keyval` for each $\langle elt \rangle$ of the $\langle keyval list \rangle$ being processed.

<code>\keys_parse:n</code>	<code>\keys_parse:n</code> $\langle keyval list \rangle$
----------------------------	--

Functions which actually parses $\langle keyval list \rangle$.

<code>\keys_parse_list:n</code>	<code>\keys_parse_list:n</code> $\langle keyval list \rangle$
---------------------------------	---

Set up macro for `\keys_parse:n`.

<code>\keys_path_add:N</code>	<code>\keys_path_add:N <tlp></code>
<code>\keys_path_add:w</code>	<code>\keys_path_add:w <tlst> \q_stop</code>

Adds a full path to key name stored in $\langle tlp \rangle$ or given as $\langle tlist \rangle$.

<code>\keys_process_elt:nn</code>	<code>\keys_process_elt:nn <elt> <value></code>
-----------------------------------	---

Lead-off processor for converting $\langle elt \rangle$ into a fully-qualified $\langle key \rangle$ and checking validity of $\langle value \rangle$.

<code>\keys_separate_path:</code>	<code>\keys_separate_path:</code>
<code>\keys_separate_path:w</code>	<code>\keys_separate_path:w / <path₁> / <path₂> \q_stop</code>

Separates key into $\langle path \rangle$ and $\langle key \text{ name} \rangle$.

<code>\keys_set:NN</code>	<code>\keys_set:NN <function> <variable></code>
<code>\keys_set:Nnn</code>	<code>\keys_set:Nnn <function> <prefix> <name></code>

Uses $\langle function \rangle$ to store $\langle value \rangle$ in $\langle variable \rangle$. If a $\langle name \rangle$ is given, it is used to construct a cname for the variable including the current $\langle module \rangle$. The $\langle prefix \rangle$ should be l_- or g_- .

<code>\keys_set_eq:nn</code>	<code>\keys_set_eq:nn <key₁> <key₂></code>
------------------------------	--

Sets $\langle key_1 \rangle$ equal to $\langle key_2 \rangle$.

<code>\keys_set_cmd:nn</code>	<code>\keys_set_cmd:nn <key> <code></code>	
<code>\keys_set_cmd:nx</code>		
<code>\keys_set_cmd:nNn</code>		<code>\keys_set_cmd:nNn <key> <number> <code></code>
<code>\keys_set_cmd:nNx</code>		<code>\keys_set_cmd:nNx <key> <number> <code></code>

Creates a $._cmd:w$ function for $\langle key \rangle$, with definition $\langle code \rangle$. The N variant can include $\langle number \rangle$ parameters.

<code>\keys_store:nn</code>	<code>\keys_store:nn <key> <data></code>
<code>\keys_store:nx</code>	

Stores $\langle data \rangle$ in $\langle key \rangle$ function.

<code>\keys_toks_set:Nn</code>	<code>\keys_toks_set:Nn <toks> <key></code>
--------------------------------	---

Sets $\langle toks \rangle$ equal to the content of $\langle key \rangle$

<code>\keys_try:</code>	<code>\keys_try:</code>
-------------------------	-------------------------

Attempt to execute a key, with no error if the key is unknown.

`\keys_undefine:n` `\keys_undefine:n <key>`
Delete definition of `<key>`.

`\keys_use:n` `\keys_use:n <key>`
Use definition of `<key>`.

`\keys_use_cmd:n` `\keys_use_cmd:n <key>`
`\keys_use_cmd:nn` `\keys_use_cmd:nn <key> <arg>`
Uses the `._cmd:w` function for `<key>`, passing `<arg>` if needed.

1.4.1 Internal properties

The internal key properties should not be accessed directly.

`._cmd:w`
The function which is executed for a key with `.code`.

`._default_tlp`
Holds the default value for a key.

`._forbidden_bool`
Indicates that a value cannot be given for `<key>`.

`._num_args_tlp`
For functions defined with `.code:Nn` and `.code:Nx`, contains the number of arguments the associated `._cmd:w` function takes.

`._required_bool`
Indicates that `<key>` must have a value provided.

1.5 Implementation

The usual preliminaries. The key–value parsing itself is handled by `l3keyval`, which does the very low-level stuff so there is no need to worry here.

```
1 {*package}
2 \ProvidesExplPackage
3 {\filename}{\filedate}{\fileversion}{\filedescription}
4 \RequirePackage{l3keyval,l3messages,l3clist,l3skip}
```

1.5.1 Variables and contrasts

`\c_keys_0_empty_tlp` A set of empty arguments.

```

5 \tlp_new:cn { c_keys_0_empty_tlp } {}
6 \tlp_new:cn { c_keys_1_empty_tlp } { {} }
7 \tlp_new:cn { c_keys_2_empty_tlp } { {} {} }
8 \tlp_new:cn { c_keys_3_empty_tlp } { {} {} {} }
9 \tlp_new:cn { c_keys_4_empty_tlp } { {} {} {} {} }
10 \tlp_new:cn { c_keys_5_empty_tlp } { {} {} {} {} {} }
11 \tlp_new:cn { c_keys_6_empty_tlp } { {} {} {} {} {} {} }
12 \tlp_new:cn { c_keys_7_empty_tlp } { {} {} {} {} {} {} {} }
13 \tlp_new:cn { c_keys_8_empty_tlp } { {} {} {} {} {} {} {} {} }
14 \tlp_new:cn { c_keys_9_empty_tlp } { {} {} {} {} {} {} {} {} {} }

```

`\c_keys_cs_prefix_tlp` First, the small number of constants needed are created. A prefix is used to keep all of the actual key macros in one place.

```
15 \tlp_new:Nn \c_keys_cs_prefix_tlp { keys-root }
```

`\c_keys_errors_path_tlp` The locations of all of the keys used by keys3 itself.

```

16 \tlp_new:Nn \c_keys_errors_path_tlp { /keys/errors }
17 \tlp_new:Nn \c_keys_properties_path_tlp { /keys/properties }
18 \tlp_new:Nn \c_keys_utilities_path_tlp { /keys }

```

`\c_keys_root_tlp` The key root should have a clear name; like all of the key macros, this does not include the prefix.

```
19 \tlp_new:Nn \c_keys_root_tlp { / }
```

`\l_keys_choice_code_tlp` When making choices, the code for each key has to be stored.

```
20 \tlp_new:N \l_keys_choice_code_tlp
```

`\l_keys_current_choice_tlp` Multiple choices need some storage.

```

21 \tlp_new:N \l_keys_current_choice_tlp
22 \int_new:N \l_keys_current_choice_int

```

`\l_keys_current_key_full_tlp` The current key name and the fully-qualified key are stored.

```

23 \tlp_new:N \l_keys_current_key_full_tlp
24 \tlp_new:N \l_keys_current_default_tlp

```

`\l_keys_current_module_tlp` The module name of the current module is stored here.

```
25 \tlp_new:N \l_keys_current_module_tlp
```

`\l_keys_current_path_tlp` The current and default paths can be stored as tpls. The default path is then initialised as the key root.

`\l_keys_default_path_tlp`

`\l_keys_choice_path_tlp`

```

26 \tlp_new:N \l_keys_current_path_tlp
27 \tlp_new:N \l_keys_default_path_tlp
28 \tlp_set_eq:NN \l_keys_default_path_tlp \c_keys_root_tlp
29 \tlp_new:N \l_keys_choice_path_tlp

```

`\l_keys_current_value_toks` The current value is stored in a token register.

```

30 \toks_new:N \l_keys_current_value_toks

```

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

```

31 \bool_new:N \l_keys_no_value_bool

```

`\l_keys_success_bool` A switch for trying keys.

```

32 \bool_new:N \l_keys_success_bool

```

`\l_keys_tmpa_tlp` A scratch area.

```

33 \tlp_new:N \l_keys_tmpa_tlp

```

1.5.2 Functions

`\keys_manage:n` The main key management macros both call the auxiliary function after setting up the parser. The expansion trick means a literal path is sent to the later function, and so the default path can be redefined.

`\keys_manage_quick:n`

`\keys_manage_internal:n`

`\keys_manage_aux:nn`

`\keys_manage_aux:Vn`

```

34 \cs_new:NNn \keys_manage:n 1 {
35   \cs_set_eq:NN \keys_parse:n \KV_parse_space_removal_sanitize:n
36   \tlp_clear:N \l_keys_current_module_tlp
37   \keys_manage_internal:n {#1}
38 }
39 \cs_new:NNn \keys_manage_quick:n 1 {
40   \let:NN \keys_parse:n \KV_parse_no_space_removal_no_sanitize:n
41   \tlp_clear:N \l_keys_current_module_tlp
42   \keys_manage_internal:n {#1}
43 }
44 \cs_new:NNn \keys_manage_internal:n 1 {
45   \keys_manage_aux:Vn \l_keys_default_path_tlp {#1}
46 }
47 \cs_new:NNn \keys_manage_aux:nn 2 {
48   \tlp_set_eq:NN \l_keys_default_path_tlp \c_keys_root_tlp
49   \keys_parse_list:n {#2}
50   \tlp_set:Nn \l_keys_default_path_tlp {#1}
51 }
52 \exp_def_form:nnn { keys_manage_aux } { nn } { Vn }

```

`\keys_manage:nn` This version uses the same tricks but includes the path as a second argument. When
`\keys_manage_quick:nn` setting lots of keys separately, this is a little faster than the key-based method.

```

\keys_manage_aux:nnn
\keys_manage_aux:Vnn
53 \cs_new:NNn \keys_manage:nn 2 {
54   \let:NN \keys_parse:n \KV_parse_space_removal_sanitize:n
55   \keys_manage_aux:Vnn \l_keys_default_path_tlp {#1} {#2}
56 }
57 \cs_new:NNn \keys_manage_quick:nn 2 {
58   \let:NN \keys_parse:n \KV_parse_space_no_removal_no_sanitize:n
59   \keys_manage_aux:Vnn \l_keys_default_path_tlp {#1} {#2}
60 }
61 \cs_new:NNn \keys_manage_aux:nnn 3 {
62   \tlp_set:Nn \l_keys_default_path_tlp {#2}
63   \tlp_clear:N \l_keys_current_module_tlp
64   \keys_parse_list:n {#3}
65   \tlp_set:Nn \l_keys_default_path_tlp {#1}
66 }
67 \cs_new:NNn \keys_manage_aux:Vnn 3 {
68   \exp_args:NV \keys_manage_aux:nnn #1 {#2} {#3}
69 }

```

1.5.3 Internal functions

`\keys_bool_new:n` To create a new switch, which will be true (as the existence of the switch is the flag here).

```

70 \cs_new:NNn \keys_bool_new:n 1 {
71   \bool_new:c { \c_keys_cs_prefix_tlp #1 }
72   \bool_set_true:c { \c_keys_cs_prefix_tlp #1 }
73 }

```

`\keys_bool_set:nN` Boolean keys are created by using the fact that only `true` and `false` give the right result
`\keys_bool_set:nnn` when looking for a setting function. A default is also set, so that the key name alone can be given.

```

74 \cs_new:NNn \keys_bool_set:nN 2 {
75   \keys_parse_list:n {
76     \c_keys_properties_path_tlp /.code:n = {
77       \cs_if_really_exist:cTF { bool_ #1 _ ##1 :N } {
78         \use:c { bool_ #1 _ ##1 :N } #2
79       }{
80         \keys_err_use:nn { boolean_expected } {##1}
81       }
82     },
83     \l_keys_current_path_tlp /.default:n = true
84   }
85 }
86 \cs_new:NNn \keys_bool_set:nnn 3 {
87   \keys_parse_list:n {
88     \c_keys_properties_path_tlp /.code:x = {

```

```

89     \exp_not:N \cs_if_really_exist:cTF { bool_ #1 _ ##1 :c } {
90       \exp_not:N \use:c { bool_ #1 _ ##1 :c }
91       { #2 \l_keys_current_module_tlp #3 _bool }
92     }{
93       \exp_not:N \keys_err_use:nn { boolean_expected } {##1}
94     }
95   },
96   \l_keys_current_path_tlp /.default:n = true
97 }
98 }

```

`\keys_bool_set_inverse:n` To set keys with reversed logic, the basics can be done in the same way as for the standard switches. The only thing needed is a reversal of true/false.

`\keys_bool_set_inverse:nN`

`\keys_bool_set_inverse:nnn`

```

99 \cs_new:NNn \keys_bool_set_inverse:n 1 {
100   \tlist_if_eq:nnTF {#1} { true } { false } { true }
101 }
102 \cs_new:NNn \keys_bool_set_inverse:nN 2 {
103   \keys_parse_list:n {
104     \c_keys_properties_path_tlp /.code:n = {
105       \cs_if_really_exist:cTF { bool_ #1 _ ##1 :N } {
106         \use:c { bool_ #1 _ \keys_bool_set_inverse:n {##1} :N } #2
107       }{
108         \keys_err_use:nn { boolean_expected } {##1}
109       }
110     },
111     \l_keys_current_path_tlp /.default:n = true
112   }
113 }
114 \cs_new:NNn \keys_bool_set_inverse:nnn 3 {
115   \keys_parse_list:n {
116     \c_keys_properties_path_tlp /.code:x = {
117       \exp_not:N \cs_if_really_exist:cTF { bool_ #1 _ ##1 :c } {
118         \exp_not:N \use:c {
119           bool_ #1 _ \exp_not:N \keys_bool_set_inverse:n {##1} :c
120         } { #2 \l_keys_current_module_tlp #3 _bool }
121       }{
122         \exp_not:N \keys_err_use:nn { boolean_expected } {##1}
123       }
124     },
125     \l_keys_current_path_tlp /.default:n = true
126   }
127 }

```

`\keys_choices_create:Nnn` When making multiple choices, the code for each choice is the same. Only the path and counter need to be altered.

`\keys_choices_create_aux:n`

`\keys_choices_create_aux:x`

`\keys_choice_create:n`

```

128 \cs_new:NNn \keys_choices_create:Nnn 3 {
129   \tlist_set_eq:NN \l_keys_choice_path_tlp \l_keys_current_path_tlp

```

```

130 \int_zero:N \l_keys_current_choice_int
131 \use:c { keys_choices_create_aux: #1 } {#3}
132 \clist_map_function:nN {#2} \keys_choice_create:n
133 \keys_manage_internal:n { \l_keys_choice_path_tlp /.expects_choice: }
134 }
135 \cs_new:NNn \keys_choices_create_aux:n 1 {
136 \tlp_set:Nn \l_keys_choice_code_tlp { \exp_not:n {#1} }
137 }
138 \cs_new:NNn \keys_choices_create_aux:x 1 {
139 \tlp_set:Nn \l_keys_choice_code_tlp {#1}
140 }
141 \cs_new:NNn \keys_choice_create:n 1 {
142 \int_incr:N \l_keys_current_choice_int
143 \keys_parse_list:n {
144 \l_keys_choice_path_tlp / #1 /.code:x = {
145 \exp_not:n { \int_set:Nn \l_keys_current_choice_int }
146 { \int_use:N \l_keys_current_choice_int }
147 \exp_not:n { \tlp_set:Nn \l_keys_current_choice_tlp } {#1}
148 \l_keys_choice_code_tlp
149 }
150 }
151 }

```

`\keys_clear_properties:n` To avoid problem on redefinition, all properties are removed.

```

152 \cs_new:NNn \keys_clear_properties:n 1 {
153 \keys_undefine:n { #1/.boolean }
154 \keys_undefine:n { #1/.cmd:w }
155 \keys_undefine:n { #1/.default_tlp }
156 \keys_undefine:n { #1/.forbidden_bool }
157 \keys_undefine:n { #1/.num_args_tlp }
158 \keys_undefine:n { #1/.required_bool }
159 }

```

`\keys_default_add:` Copies the default value to the current one if needed.

```

160 \cs_new:NNn \keys_default_add: 0 {
161 \bool_if:NT \l_keys_no_value_bool {
162 \keys_if_really_exist:nT
163 { \l_keys_current_key_full_tlp /.default_tlp } {
164 \keys_toks_set:Nn \l_keys_current_value_toks
165 { \l_keys_current_key_full_tlp /.default_tlp }
166 \bool_set_false:N \l_keys_no_value_bool
167 }
168 }
169 }

```

`\keys_find_code_full:` Two functions to find something to process the key value given. First, a search is made
`\keys_find_code_name:` for either a command property for the key, or a function for the key itself. If that fails, the generic handlers are used after separating out the key name and key path.

```

170 \cs_new:NNn \keys_find_code_full: 0 {
171   \keys_if_cmd_really_exist:nTF { \l_keys_current_key_full_tlp } {
172     \keys_use_cmd:n { \l_keys_current_key_full_tlp }
173   }{
174     \keys_if_really_exist:nTF { \l_keys_current_key_full_tlp } {
175       \bool_if:NTF \l_keys_no_value_bool {
176         \keys_use:n { \l_keys_current_key_full_tlp }
177       }{
178         \keys_store:nx { \l_keys_current_key_full_tlp }
179         { \toks_use:N \l_keys_current_value_toks }
180       }
181     }{
182       \keys_find_code_name:
183     }
184   }
185 }
186 \cs_new:NNn \keys_find_code_name: 0 {
187   \keys_separate_path:
188   \keys_if_cmd_really_exist:nTF
189     { \c_keys_properties_path_tlp / \l_keys_current_key_name_tlp } {
190     \keys_use_cmd:n
191       { \c_keys_properties_path_tlp / \l_keys_current_key_name_tlp }
192   }{
193     \keys_if_cmd_really_exist:nTF
194     { \l_keys_current_path_tlp / unknown } {
195     \keys_use_cmd:n { \l_keys_current_path_tlp / unknown }
196   }{
197     \keys_err_use:n { unknown_key }
198   }
199 }
200 }

```

`\keys_if_cmd_really_exist:nTF` A dedicated check for the `._cmd:w` property key.

```

201 \def_long_test_function_new:npn { keys_if_cmd_really_exist:n } #1 {
202   \if_cs_exist:w \c_keys_cs_prefix_tlp #1 /._cmd:w \cs_end:
203 }

```

`\keys_if_really_exist:nTF` Check if a key exists without adding to the hash table.

```

204 \def_long_test_function_new:npn { keys_if_really_exist:n } #1 {
205   \if_cs_exist:w \c_keys_cs_prefix_tlp #1 \cs_end:
206 }

```

`\keys_if_value:nTF` Used for required and forbidden values.

```

207 \def_long_test_function_new:npn { keys_if_value:n } #1 {
208   \if_cs_exist:w
209     \c_keys_cs_prefix_tlp \l_keys_current_key_full_tlp /._ #1 _bool
210   \cs_end:
211 }

```

`\keys_no_value_elt:n` The two functions passed to `l3keyval` to actually act on each key found.

`\keys_value_elt:nn`

```
212 \cs_new:NNn \keys_no_value_elt:n 1 {
213   \bool_set_true:N \l_keys_no_value_bool
214   \keys_process_elt:nn {#1} { }
215 }
216 \cs_new:NNn \keys_value_elt:nn 2 {
217   \bool_set_false:N \l_keys_no_value_bool
218   \keys_process_elt:nn {#1} {#2}
219 }
```

`\keys_parse:n` The macro used to actually process the key–value input is taken from `l3keyval`. There are two possible options, and so at this stage the macro is simply reserved.

```
220 \cs_new:NNn \keys_parse:n 1 {\ERROR}
```

`\keys_parse_list:n` All of the management macros call this common parser. First, the key processing macros are defined, then the appropriate parser is called.

```
221 \cs_new:NNn \keys_parse_list:n 1 {
222   \let:NN \KV_key_value_elt:nn \keys_value_elt:nn
223   \let:NN \KV_key_no_value_elt:n \keys_no_value_elt:n
224   \keys_parse:n {#1}
225 }
```

`\keys_path_add:N` The code to check for a path looks for a `/` at the start of the key.

`\keys_path_add:w`

`\keys_path_add_aux:w`

```
226 \cs_new:NNn \keys_path_add:N 1 {
227   \exp_after:NN \keys_path_add:w #1 \q_stop
228 }
229 \cs_new:Npn \keys_path_add:w {
230   \exp_after:NN \peek_meaning:NTF \c_keys_root_tlp {
231     \use_none_delimit_by_q_stop:w
232   }{
233     \keys_path_add_aux:w
234   }
235 }
236 \cs_new:Npn \keys_path_add_aux:w #1 \q_stop {
237   \tlp_set:Nx \l_keys_current_key_full_tlp {\l_keys_default_path_tlp #1}
238 }
```

`\keys_process_elt:nn` The key processor starts by storing the given key name and value, and adding a path to the former if necessary. There is then potentially a need to fill in a default value before checking for required or forbidden values.

`\keys_process_elt_aux:`

```
239 \cs_new:NNn \keys_process_elt:nn 2 {
240   \tlp_set:Nx \l_keys_current_key_full_tlp {#1}
241   \toks_set:Nn \l_keys_current_value_toks {#2}
242   \keys_path_add:N \l_keys_current_key_full_tlp
```

```

243 \keys_default_add:
244 \keys_if_value:nTF { required } {
245   \bool_if:NTF \l_keys_no_value_bool {
246     \keys_err_use:n { value_required }
247   }{
248     \keys_process_elt_aux:
249   }
250 }{
251   \keys_process_elt_aux:
252 }
253 }
254 \cs_new:NNn \keys_process_elt_aux: 0 {
255   \keys_if_value:nTF { forbidden } {
256     \bool_if:NTF \l_keys_no_value_bool {
257       \keys_find_code_full:
258     }{
259       \keys_err_use:nn { value_forbidden }
260       { \toks_use:N \l_keys_current_value_toks }
261     }
262   }{
263     \keys_find_code_full:
264   }
265 }

```

`\keys_separate_path:` A simple piece of recursion to find the key name and path.

`\keys_separate_path:w`

```

266 \cs_new:NNn \keys_separate_path: 0 {
267   \tlp_clear:N \l_keys_current_path_tlp
268   \exp_after:NN \keys_separate_path:w \l_keys_current_key_full_tlp
269   / \q_stop
270 }
271 \cs_new:Npn \keys_separate_path:w / #1 / #2 \q_stop {
272   \tlist_if_empty:nTF {#2} {
273     \tlp_set:Nn \l_keys_current_key_name_tlp {#1}
274   }{
275     \tlp_put_right:Nn \l_keys_current_path_tlp { / #1 }
276     \keys_separate_path:w /#2 \q_stop
277   }
278 }

```

`\keys_set:NN` Storage of data in outside of keys.

`\keys_set:Nnn`

```

279 \cs_new:NNn \keys_set:NN 2 {
280   \keys_parse_list:n {
281     \c_keys_properties_path_tlp /.code:n = {
282       #1 #2 {##1}
283     }
284   }
285 }
286 \cs_new:NNn \keys_set:Nnn 3 {

```

```

287 \keys_parse_list:n {
288   \c_keys_properties_path_tlp /.code:x = {
289     \exp_not:N #1 { #2 \l_keys_current_module_tlp #3 } {##1}
290   }
291 }
292 }

```

`\keys_set_eq:nn` To alias one key to another.

```

293 \cs_new:NNn \keys_set_eq:nn 2 {
294   \cs_set_eq:cc { \c_keys_cs_prefix_tlp #1 }
295   { \c_keys_cs_prefix_tlp #2 }
296 }

```

`\keys_set_cmd:nn` Creation of the key `._cmd:w` macros happens here. For the multiple-argument versions, the number of arguments is stored for use later on.

`\keys_set_cmd:nx`

`\keys_set_cmd:nNn`

`\keys_set_cmd:nNx`

`._cmd:w`

`._num_args_tlp`

```

297 \cs_new:NNn \keys_set_cmd:nn 2 {
298   \keys_clear_properties:n {#1}
299   \cs_set:cNn { \c_keys_cs_prefix_tlp #1 /._cmd:w } 1 {#2}
300 }
301 \cs_new:NNn \keys_set_cmd:nx 2 {
302   \keys_clear_properties:n {#1}
303   \cs_set:cNx { \c_keys_cs_prefix_tlp #1 /._cmd:w } 1 {#2}
304 }
305 \cs_new:NNn \keys_set_cmd:nNn 3 {
306   \keys_clear_properties:n {#1}
307   \cs_set:cNn { \c_keys_cs_prefix_tlp #1 /._cmd:w } #2 {#3}
308   \keys_store:nn { #1 /._num_args_tlp } {#2}
309 }
310 \cs_new:NNn \keys_set_cmd:nNx 3 {
311   \keys_clear_properties:n {#1}
312   \cs_set:cNx { \c_keys_cs_prefix_tlp #1 /._cmd:w } #2 {#3}
313   \keys_store:nn { #1 /._num_args_tlp } {#2}
314 }

```

`\keys_store:nn` Direct storage of data in keys.

`\keys_store:nx`

```

315 \cs_new:NNn \keys_store:nn 2 {
316   \tlp_set:cn { \c_keys_cs_prefix_tlp #1 } {#2}
317 }
318 \cs_new:NNn \keys_store:nx 2 {
319   \tlp_set:cx { \c_keys_cs_prefix_tlp #1 } {#2}
320 }

```

`\keys_toks_set:Nn` Sets a toks to the content of a key.

```

321 \cs_new:NNn \keys_toks_set:Nn 2 {
322   \exp_args:NNv \toks_set:Nn #1 { \c_keys_cs_prefix_tlp #2 }
323 }

```

`\keys_try:` A function to look for the `._cmd:w` property of a key, and executes it if found.

```

324 \cs_new:NNn \keys_try: 0 {
325   \bool_set_false:N \l_keys_success_bool
326   \keys_if_cmd_really_exist:nT { \l_keys_current_path_tlp } {
327     \bool_set_true:N \l_keys_success_bool
328     \bool_if:NT \l_keys_no_value_bool {
329       \keys_if_really_exist:nT
330         { \l_keys_current_path_tlp /._default_tlp } {
331           \keys_toks_set:Nn \l_keys_current_value_toks
332             { \l_keys_current_path_tlp /._default_tlp }
333         }
334     }
335     \let:NN \l_keys_current_key_full_tlp \l_keys_current_path_tlp
336     \keys_use_cmd:n { \l_keys_current_path_tlp }
337   }
338 }

```

`\keys_undefine:n` To remove a key.

```

339 \cs_new:NNn \keys_undefine:n 1 {
340   \cs_set_eq:cN { \c_keys_cs_prefix_tlp #1 } \c_undefined
341 }

```

`\keys_use:n` Use whatever is stored in a key.

```

342 \cs_new:NNn \keys_use:n 1 {
343   \use:c { \c_keys_cs_prefix_tlp #1 }
344 }

```

`\keys_use_cmd:n` Some care is needed when using command keys. For commands with multiple arguments a check is made in case none were given, and if so a series of empty values is given instead.

`\keys_use_cmd:nn`
`\keys_use_cmd_aux:nn`
`\keys_use_cmd_aux:w`

```

345 \cs_new:NNn \keys_use_cmd:n 1 {
346   \exp_args:Nno \keys_use_cmd:nn {#1}
347   { \toks_use:N \l_keys_current_value_toks }
348 }
349 \cs_new:NNn \keys_use_cmd:nn 2 {
350   \keys_if_really_exist:nTF { #1 /._num_args_tlp } {
351     \keys_use_cmd_aux:nn {#1} {#2}
352   }{
353     \keys_use:n {#1/._cmd:w} {#2}
354   }
355 }
356 \cs_new:NNn \keys_use_cmd_aux:nn 2 {
357   \tlist_if_empty:nTF {#2} {
358     \cs_set:NNn \keys_use_cmd_aux:w 0 {
359       \keys_use:n { #1 /._cmd:w }
360     }

```

```

361     \exp_after:NN \exp_after:NN \exp_after:NN \keys_use_cmd_aux:w
362     \cs:w
363     c_keys_ \keys_use:n { #1 /._num_args_tlp } _empty_tlp
364     \cs_end:
365 }{
366     \keys_use:n { #1 /._cmd:w } #2
367 }
368 }

```

1.5.4 Error handling code

The L^AT_EX3 approach is to have named errors called separately and defined separately. To make life a little easier here, some custom functions are used to keep repetition down.

`\keys_err_new:nNnnn` To create new error messages, a utility function is created.

```

369 \cs_new:NNn \keys_err_new:nNnnn 1 {
370     \tlp_new:cn { l_keys_err_#1_tlp } {#1}
371     \exp_args:NNc \err_interrupt_new:NNNnnn \c_keys_err_tlp
372     { l_keys_err_#1_tlp }
373 }

```

`\l_keys_err_unknown_key_tlp` Text for package error messages is stored in `keys3.err`. The error names do not need
`\l_keys_err_value_ignored_tlp` `\keys` at the start as they have to be in the file.
`\l_keys_err_value_required_tlp`

```

374 \err_file_new:Nn \c_keys_err_tlp { keys3.err }
375 \keys_err_new:nNnnn { unknown_key } 1
376 { The-key-‘#1’-is-unknown-and-is-being-ignored }
377 { \err_help_return_or_X: } { }
378 \keys_err_new:nNnnn { value_required } 1
379 { The-key-‘#1’-requires-a-value-and-is-being-ignored }
380 { \err_help_return_or_X: } { }
381 \keys_err_new:nNnnn { value_forbidden } 2
382 { The-key-‘#1’-cannot-taken-a-value:-the-given-input-~\iow_newline:
383     ~\text_put_sp:-\text_put_sp:‘#2’-is-being-ignored }
384 { \err_help_return_or_X: } { }
385 \keys_err_new:nNnnn { boolean_expected } 2
386 { Key-‘#1’-takes-the-Boolean-values-‘true’-and-~\iow_newline:~
387     \text_put_sp:-\text_put_sp:‘false’-only.-The-given-value-‘#2’-is-
388     being-ignored }
389 { \err_help_return_or_X: } { }
390 \keys_err_new:nNnnn { not_boolean } 1
391 { Key-‘#1’-is-not-a-Boolean-key:-you-cannot-create-a-complement }
392 { \err_help_return_or_X: } { }
393 \keys_err_new:nNnnn { unknown_choice } 2
394 { Choice-‘#2’-unknown-for-key-‘#1’:~\iow_newline:~\text_put_sp:~
395     \text_put_sp:-the-key-is-being-ignored }
396 { \err_help_return_or_X: } { }
397 \err_file_close:N \c_keys_err_tlp

```

`\keys_err_use:nw` Utilities for using errors: the first function is `nw` as it may need one or two arguments in addition to the `n`.

```

\keys_err_use:n
\keys_err_use:nn
398 \cs_new:NNn \keys_err_use:nw 1 {
399   \exp_args:NNc \err_interrupt:NNw \c_keys_err_tlp { l_keys_err_#1_tlp }
400 }
401 \cs_new:NNn \keys_err_use:n 1 {
402   \keys_err_use:nw {#1} \l_keys_current_key_full_tlp
403 }
404 \cs_new:NNn \keys_err_use:nn 2 {
405   \keys_err_use:nw {#1} \l_keys_current_key_full_tlp {#2}
406 }

```

1.5.5 Property definitions

`.code:n` The `.code:n` and `.code:Nn` properties have to be defined directly.

```

.code:Nn
407 \keys_set_cmd:nn { \c_keys_properties_path_tlp /.code:n } {
408   \keys_set_cmd:nn { \l_keys_current_path_tlp } {#1}
409 }
410 \keys_set_cmd:nNn { \c_keys_properties_path_tlp /.code:Nn } 2 {
411   \keys_set_cmd:nNn { \l_keys_current_path_tlp } #1 {#2}
412 }

```

The remaining definitions can all be carried out using the package itself. As category codes and spaces are not an issue here, the `_quick` version of `\keys_manage` is used.

First, an error is created for unknown keys: this is done early to catch any internal errors.

```

413 \keys_manage_quick:n {
414   \c_keys_errors_path_tlp /unknown/.code:n = {
415     \keys_err_use:n { unknown_key }
416   }
417 }

```

`.code:x` Fully-expanded versions of the basic `.code` properties.

```

.code:Nx
418 \keys_manage_quick:n {
419   \c_keys_properties_path_tlp /.code:x/.code:n = {
420     \keys_set_cmd:nx { \l_keys_current_path_tlp } {#1}
421   },
422   \c_keys_properties_path_tlp /.code:Nx/.code:Nn = 2 {
423     \keys_set_cmd:nNx { \l_keys_current_path_tlp } {#1} {#2}
424   }
425 }

```

`.cd:` The change-directory property simply alters the value of the default path.

```

426 \keys_manage_quick:n {

```

```

427     \c_keys_properties_path_tlp /.cd:/.code:n = {
428     \tlp_set:Nx \l_keys_default_path_tlp { \l_keys_current_path_tlp / }
429     }
430 }

```

`.value_forbidden:` Values are required or forbidden by creating the appropriate flags.

```

    ._required_bool
    .value_required:
    ._forbidden_bool
431 \keys_manage_quick:n {
432   \c_keys_properties_path_tlp /.value_required:/.code:n = {
433     \keys_bool_new:n { \l_keys_current_path_tlp /.required_bool }
434     \keys_undefine:n { \l_keys_current_path_tlp /.forbidden_bool }
435   },
436   \c_keys_properties_path_tlp /.value_forbidden:/.code:n = {
437     \keys_bool_new:n { \l_keys_current_path_tlp /.forbidden_bool }
438     \keys_undefine:n { \l_keys_current_path_tlp /.required_bool }
439   }
440 }

```

`.default:n` The default value for a key is stored in the `._default_tlp` private property.

```

    ._default_tlp
441 \keys_manage_quick:n {
442   \c_keys_properties_path_tlp /.default:n/.code:n = {
443     \keys_store:nn { \l_keys_current_path_tlp /.default_tlp } {#1}
444     \keys_undefine:n { \l_keys_current_path_tlp /.required_bool }
445   }
446 }

```

`.tlp_set:N` Storage of the value in a tlp, as given or expanded, local or global.

```

    .tlp_set_x:N
    .tlp_gset:N
    .tlp_gset_x:N
447 \keys_manage_quick:n {
448   \c_keys_properties_path_tlp /.tlp_set:N/.code:n = {
449     \keys_set:NN \tlp_set:Nn #1
450   },
451   \c_keys_properties_path_tlp /.tlp_set_x:N/.code:n = {
452     \keys_set:NN \tlp_set:Nx #1
453   },
454   \c_keys_properties_path_tlp /.tlp_gset:N/.code:n = {
455     \keys_set:NN \tlp_gset:Nn #1
456   },
457   \c_keys_properties_path_tlp /.tlp_gset_x:N/.code:n = {
458     \keys_set:NN \tlp_gset:Nx #1
459   }
460 }

```

`.int_set:N` For int, skip and toks storage, no expansion to worry about.

```

    .int_gset:N
    .skip_set:N
    .skip_gset:N
    .toks_set:N
    .toks_gset:N
461 \keys_manage_quick:n {
462   \c_keys_properties_path_tlp /.int_set:N/.code:n = {
463     \keys_set:NN \int_set:Nn #1

```

```

464 },
465 \c_keys_properties_path_tlp /.int_gset:N/.code:n = {
466   \keys_set:NN \int_gset:Nn #1
467 },
468 \c_keys_properties_path_tlp /.skip_set:N/.code:n = {
469   \keys_set:NN \skip_set:Nn #1
470 },
471 \c_keys_properties_path_tlp /.skip_gset:N/.code:n = {
472   \keys_set:NN \skip_gset:Nn #1
473 },
474 \c_keys_properties_path_tlp /.toks_set:N/.code:n = {
475   \keys_set:NN \toks_set:Nn #1
476 },
477 \c_keys_properties_path_tlp /.toks_gset:N/.code:n = {
478   \keys_set:NN \toks_gset:Nn #1
479 }
480 }

```

The prefix for the current module is needed when storing material by csname.

```

481 \keys_manage_quick:n {
482   \c_keys_utilities_path_tlp /current_module:n/.code:n = {
483     \tlist_if_empty:nTF {#1} {
484       \tlp_clear:N \l_keys_current_module_tlp
485     }{
486       \tlp_set:Nn \l_keys_current_module_tlp { #1 _ }
487     }
488   }
489 }

```

.tlp_set:n With the module available, storage properties that only need the unique part of the variable name are created.

```

.tlp_set_x:n
.tlp_gset:n
.tlp_gset_x:n
.int_set:n
.int_gset:n
.skip_set:n
.skip_gset:n
.toks_set:n
.toks_gset:n
490 \keys_manage_quick:n {
491   \c_keys_properties_path_tlp /.tlp_set:n/.code:n = {
492     \keys_set:Nnn \tlp_set:cn { l_ } { #1 _tlp }
493   },
494   \c_keys_properties_path_tlp /.tlp_set_x:n/.code:n = {
495     \keys_set:Nnn \tlp_set:cx { l_ } { #1 _tlp }
496   },
497   \c_keys_properties_path_tlp /.tlp_gset:n/.code:n = {
498     \keys_set:Nnn \tlp_gset:cn { g_ } { #1 _tlp }
499   },
500   \c_keys_properties_path_tlp /.tlp_gset_x:n/.code:n = {
501     \keys_set:Nnn \tlp_gset:cx { g_ } { #1 _tlp }
502   },
503   \c_keys_properties_path_tlp /.int_set:n/.code:n = {
504     \keys_set:Nnn \int_set:cn { l_ } { #1 _int }
505   },

```

```

506 \c_keys_properties_path_tlp /.int_gset:n/.code:n = {
507   \keys_set:Nnn \int_gset:cn { g_ } { #1 _int }
508 },
509 \c_keys_properties_path_tlp /.skip_set:n/.code:n = {
510   \keys_set:Nnn \skip_set:cn { l_ } { #1 _skip }
511 },
512 \c_keys_properties_path_tlp /.skip_gset:n/.code:n = {
513   \keys_set:Nnn \skip_gset:cn { g_ } { #1 _skip }
514 },
515 \c_keys_properties_path_tlp /.toks_set:n/.code:n = {
516   \keys_set:Nnn \toks_set:cn { l_ } { #1 _toks }
517 },
518 \c_keys_properties_path_tlp /.toks_gset:n/.code:n = {
519   \keys_set:Nnn \toks_gset:cn { g_ } { #1 _toks }
520 },
521 }

```

.bool_set:N The properties for switches look similar, although internally things are rather different.

```

522 \keys_manage_quick:n {
523   \c_keys_properties_path_tlp /.bool_set:N/.code:n = {
524     \keys_bool_set:nN { set } #1
525   },
526   \c_keys_properties_path_tlp /.bool_gset:N/.code:n = {
527     \keys_bool_set:nN { gset } #1
528   },
529   \c_keys_properties_path_tlp /.bool_set:n/.code:n = {
530     \keys_bool_set:nnn { set } { l_ } { #1 }
531   },
532   \c_keys_properties_path_tlp /.bool_gset:n/.code:n = {
533     \keys_bool_set:nnn { gset } { g_ } { #1 }
534   },
535 }

```

.bool_set_inverse:N The inverse versions of the switches are handled in the same way here.

```

536 \keys_manage_quick:n {
537   \c_keys_properties_path_tlp /.bool_set_inverse:N/.code:n = {
538     \keys_bool_set_inverse:nN { set } #1
539   },
540   \c_keys_properties_path_tlp /.bool_gset_inverse:N/.code:n = {
541     \keys_bool_set_inverse:nN { gset } #1
542   },
543   \c_keys_properties_path_tlp /.bool_set_inverse:n/.code:n = {
544     \keys_bool_set_inverse:nnn { set } { l_ } { #1 }
545   },
546   \c_keys_properties_path_tlp /.bool_gset_inverse:n/.code:n = {
547     \keys_bool_set_inverse:nnn { gset } { g_ } { #1 }
548   },
549 }

```

`.expects_choice:` Multiple choice keys are created by searching sub-keys. So the code to make a key into a multiple choice is quite simple.

```

550 \keys_manage_quick:n {
551   \c_keys_properties_path_tlp /.expects_choice:/.code:n = {
552     \keys_manage_internal:n {
553       \l_keys_current_path_tlp /.cd:,
554       .code:n = {
555         \tlp_set:Nn \l_keys_current_choice_tlp {##1}
556         \int_zero:N \l_keys_current_choice_int
557         \exp_args:No \keys_parse_list:n
558         { \l_keys_current_key_full_tlp / ##1 }
559       },
560       unknown/.code:n = {
561         \keys_err_use:nn { unknown_choice }
562         { \l_keys_current_key_name_tlp }
563       }
564     }
565   },
566 }

```

`.create_choices:nn` Creating choices as a block.
`.create_choices:nx`

```

567 \keys_manage_quick:n {
568   \c_keys_properties_path_tlp /.create_choices:nn/.code:Nn = 2 {
569     \keys_choices_create:Nnn n {#1} {#2}
570   },
571   \c_keys_properties_path_tlp /.create_choices:nx/.code:Nn = 2 {
572     \keys_choices_create:Nnn x {#1} {#2}
573   },
574 }

```

`.use_keys:n` Keys calling other keys is actually quite easy.
`.use_keys:x`
`.use_keys:Nn`
`.use_keys:Nx`

```

575 \keys_manage_quick:n {
576   \c_keys_properties_path_tlp /.use_keys:n/.code:n = {
577     \keys_manage_internal:n {
578       \l_keys_current_path_tlp /.code:n = { \keys_parse_list:n {#1} }
579     }
580   },
581   \c_keys_properties_path_tlp /.use_keys:x/.code:n = {
582     \keys_manage_internal:n {
583       \l_keys_current_path_tlp /.code:x = {
584         \exp_not:N \keys_parse_list:n {#1}
585       }
586     }
587   },
588   \c_keys_properties_path_tlp /.use_keys:Nn/.code:Nn = 2 {
589     \keys_manage_internal:n {
590       \l_keys_current_path_tlp /.code:Nn = #1 { \keys_parse_list:n {#2} }

```

```

591     }
592   },
593   \c_keys_properties_path_tlp /.use_keys:Nx/.code:Nn = 2 {
594     \keys_manage_internal:n {
595       \l_keys_current_path_tlp /.code:Nx =
596       #1 { \exp_not:N \keys_parse_list:n {#2} }
597     }
598   }
599 }

```

`.try:n` For attempting to set keys without assuming they exist.

```

.retry:n
600 \keys_manage_quick:n {
601   \c_keys_properties_path_tlp /.try:n/.code:n = { \keys_try: },
602   \c_keys_properties_path_tlp /.retry:n/.code:n = {
603     \bool_if:NF \l_keys_success_bool { \keys_try: }
604   }
605 }

```

`.show_code:` Finally, two keys for debugging problems.

```

.show_key:
606 \keys_manage_quick:n {
607   \c_keys_properties_path_tlp /.show_code:/.code:n = {
608     \cs_show:c {
609       \c_keys_cs_prefix_tlp \l_keys_current_path_tlp /._cmd:w
610     }
611   },
612   \c_keys_properties_path_tlp /.show_key:/.code:n = {
613     \cs_show:c { \c_keys_cs_prefix_tlp \l_keys_current_path_tlp }
614   }
615 }
616 </package>

```