

The `lttemplates.dtx` code*

Frank Mittelbach, Chris Rowley, David Carlisle, L^AT_EX Project[†]

May 19, 2025

1 Introduction

There are three broad “layers” between putting down ideas into a source file and ending up with a typeset document. These layers of document writing are

1. authoring of the text with mark-up;
2. document layout design;
3. implementation (with T_EX programming) of the design.

We write the text as an author, and we see the visual output of the design after the document is generated; the T_EX implementation in the middle is the glue between the two.

L^AT_EX’s greatest success has been to standardise a system of mark-up that balances the trade-off between ease of reading and ease of writing to suit almost all forms of technical writing. It’s other original strength was a good background in typographical design; while the standard L^AT_EX 2_ε classes look somewhat dated now in terms of their visual design, their typography is generally sound (barring the occasional minor faults).

However, L^AT_EX 2_ε has always lacked a standard approach to customising the visual design of a document. Changing the looks of the standard classes involved either:

- Creating a new version of the implementation code of the class and editing it.
- Loading one of the many packages to customise certain elements of the standard classes.
- Loading a completely different document class, such as KOMA-Script or memoir, that allows easy customization.

All three of these approaches have their drawbacks and learning curves.

The idea behind `lttemplates` is to cleanly separate the three layers introduced at the beginning of this section, so that document authors who are not programmers can easily change the design of their documents. `lttemplates` also makes it easier for L^AT_EX programmers to provide their own customizations on top of a pre-existing class.

*This file has version v1.0e dated 2025-01-20, © L^AT_EX Project.

[†]E-mail: latex-team@latex-project.org

2 What is a document?

Besides the textual content of the words themselves, the source file of a document contains mark-up elements that add structure to the document. These elements include sectional divisions, figure/table captions, lists of various sorts, theorems/proofs, and so on. The list will be different for every document that can be written.

Each element can be represented logically without worrying about the formatting, with mark-up such as `\section`, `\caption`, `\begin{enumerate}` and so on. The output of each one of these document elements will be a typeset representation of the information marked up, and the visual arrangement and design of these elements can vary widely in producing a variety of desired outcomes.

For each type of document element, there may be design variations that contain the same sort of information but present it in slightly different ways. For example, the difference between a numbered and an unnumbered section, `\section` and `\section*`, or the difference between an itemized list or an enumerated list.

There are three distinct layers in the definition of “a document” at this level

1. semantic elements such as the ideas of sections and lists;
2. a set of design solutions for representing these elements visually;
3. specific variations for these designs that represent the elements in the document.

In the parlance of the template system, these are called types, templates, and instances, and they are discussed below in sections 4, 5, and 7, respectively.

3 Types, templates, and instances

By formally declaring documents to be composed of mark-up elements grouped into types, which are interpreted and typeset with a set of templates, each of which has one or more instances with which to compose each and every semantic unit of the text, we can cleanly separate the components of document construction.

All of the structures provided by the template system are global, and do not respect \TeX grouping.

4 Template types

An *template type* (sometimes just “type”) is an abstract idea of a document element that takes a fixed number of arguments corresponding to the information from the document author that it is representing. A sectioning type, for example, might take three inputs: “title”, “short title”, and “label”.

Any given document class will define which types are to be used in the document, and any template of a given type can be used to generate an instance for the type. (Of course, different templates will produce different typeset representations, but the underlying content will be the same.)

<code>\NewTemplateType</code>	<code>\NewTemplateType {<template type>} {<no. of args>}</code>
-------------------------------	---

This function defines an `<template type>` taking `<number of arguments>`, where the `<type>` is an abstraction as discussed above. For example,

`\NewTemplateType{sectioning}{3}`

creates a type “sectioning”, where each use of that type will need three arguments.

5 Templates

A *template* is a generalized design solution for representing the information of a specified type. Templates that do the same thing, but in different ways, are grouped together by their type and given separate names. There are two important parts to a template:

- the parameters it takes to vary the design it is producing;
- the implementation of the design.

As a document author or designer does not care about the implementation but rather only the interface to the template, these two aspects of the template definition are split into two independent declarations, `\DeclareTemplateInterface` and `\DeclareTemplateCode`.

<code>\DeclareTemplateInterface</code>	<code>\DeclareTemplateInterface</code> <code>{<type>} {<template>} {<no. of args>}</code> <code>{<key list>}</code>
--	---

A `<template>` interface is declared for a particular `<type>`, where the `<number of arguments>` must agree with the type declaration. The interface itself is defined by the `<key list>`, which is itself a key–value list taking a specialized format:

`<key1> : <key type1> ,`
`<key2> : <key type2> ,`
`<key3> : <key type3> = <default3> ,`
`<key4> : <key type4> = <default4> ,`
`...`

Each `<key>` name should consist of ASCII characters, with the exception of `,`, `=` and `□`. The recommended form for key names is to use lower case letters, with dashes to separate out different parts. Spaces are ignored in key names, so they can be included or missed out at will. Each `<key>` must have a `<key type>`, which defined the type of input that the `<key>` requires. A full list of key types is given in Table 1. Each key may have a `<default>` value, which will be used in by the template if the `<key>` is not set explicitly. The `<default>` should be of the correct form to be accepted by the `<key type>` of the `<key>`: this is not checked by the code. Expressions for numerical values are evaluated when the template is used, thus for example values given in terms of `em` or `ex` will be set respecting the prevailing font.

Key-type	Description of input
<code>boolean</code>	<code>true</code> or <code>false</code>
<code>choice{⟨choices⟩}</code>	A list of pre-defined <code>⟨choices⟩</code>
<code>commalist</code>	A comma-separated list
<code>function{⟨N⟩}</code>	A function definition with N arguments (N from 0 to 9)
<code>instance{⟨name⟩}</code>	An instance of type <code>⟨name⟩</code>
<code>integer</code>	An integer or integer expression
<code>length</code>	A fixed length
<code>muskip</code>	A math length with shrink and stretch components
<code>real</code>	A real (floating point) value
<code>skip</code>	A length with shrink and stretch components
<code>tokenlist</code>	A token list: any text or commands

Table 1: Key-types for defining template interfaces with `\DeclareTemplateInterface`.

`\KeyValue` `\KeyValue {⟨key name⟩}`

There are occasions where the default (or value) for one key should be taken from another. The `\KeyValue` function can be used to transfer this information without needing to know the internal implementation of the key:

```

\DeclareTemplateInterface { type } { template } { no. of args }
{
  key-name-1 : key-type = value ,
  key-name-2 : key-type = \KeyValue { key-name-1 },
  ...
}

```

Key-type	Description of binding
<code>boolean</code>	Boolean variable, <i>e.g.</i> <code>\l_tmpa_bool</code>
<code>choice</code>	List of choice implementations (see Section 6)
<code>commalist</code>	Comma list, <i>e.g.</i> <code>\l_tmpa_clist</code>
<code>function</code>	Function taking N arguments, <i>e.g.</i> <code>\use_i:nn</code>
<code>instance</code>	
<code>integer</code>	Integer variable, <i>e.g.</i> <code>\l_tmpa_int</code>
<code>length</code>	Dimension variable, <i>e.g.</i> <code>\l_tmpa_dim</code>
<code>muskip</code>	Muskip variable, <i>e.g.</i> <code>\l_tmpa_muskip</code>
<code>real</code>	Floating-point variable, <i>e.g.</i> <code>\l_tmpa_fp</code>
<code>skip</code>	Skip variable, <i>e.g.</i> <code>\l_tmpa_skip</code>
<code>tokenlist</code>	Token list variable, <i>e.g.</i> <code>\l_tmpa_tl</code>

Table 2: Bindings required for different key types when defining template implementations with `\DeclareTemplateCode`. Apart from `code`, `choice` and `function` all of these accept the key word `global` to carry out a global assignment.

<code>\DeclareTemplateCode</code>	<code>\DeclareTemplateCode</code> $\langle type \rangle$ $\langle template \rangle$ $\langle no. of args \rangle$ $\langle key bindings \rangle$ $\langle code \rangle$
-----------------------------------	---

The relationship between a templates keys and the internal implementation is created using the `\DeclareTemplateCode` function. As with `\DeclareTemplateInterface`, the $\langle template \rangle$ name is given along with the $\langle type \rangle$ and $\langle number of arguments \rangle$ required. The $\langle key bindings \rangle$ argument is a key–value list which specifies the relationship between each $\langle key \rangle$ of the template interface with an underlying $\langle variable \rangle$.

```

 $\langle key1 \rangle = \langle variable1 \rangle,$ 
 $\langle key2 \rangle = \langle variable2 \rangle,$ 
 $\langle key3 \rangle = global \langle variable3 \rangle,$ 
 $\langle key4 \rangle = global \langle variable4 \rangle,$ 
...

```

With the exception of the `choice`, `code` and `function` key types, the $\langle variable \rangle$ here should be the name of an existing L^AT_EX3 register. As illustrated, the key word “global” may be included in the listing to indicate that the $\langle variable \rangle$ should be assigned globally. A full list of variable bindings is given in Table 2.

The $\langle code \rangle$ argument of `\DeclareTemplateCode` is used as the replacement text for the template when it is used, either directly or as an instance. This may therefore accept arguments `#1`, `#2`, *etc.* as detailed by the $\langle number of arguments \rangle$ taken by the type.

<code>\AssignTemplateKeys</code>	<code>\AssignTemplateKeys</code>
----------------------------------	----------------------------------

In the final argument of `\DeclareTemplateCode` the assignment of keys defined by the template may be delayed by including the command `\AssignTemplateKeys`. If this is *not* present, keys are assigned immediately before the template code. If an `\AssignTemplateKeys` command is present, assignment is delayed until this point. Note that the command must be *directly* present in the code, not placed within a nested command/macro.

<code>\SetKnownTemplateKeys</code>	<code>\SetKnownTemplateKeys {<type>} {<template>} {<keyvals>}</code>
<code>\SetTemplateKeys</code>	<code>\SetTemplateKeys {<type>} {<template>} {<keyvals>}</code>
<code>\UnusedTemplateKeys</code>	<code>\UnusedTemplateKeys % all <keyvals> unused by previous \SetKnownTemplateKeys</code>

In the final argument of `\DeclareTemplateCode` one can also overwrite (some of) the current template key value settings by using the command `\SetKnownTemplateKeys` or `\SetTemplateKeys`, i.e., they can overwrite the template default values and the values assigned by the instance.

The `\SetKnownTemplateKeys` and `\SetTemplateKeys` commands are only supported within the code of a template; using them elsewhere has unpredictable results. If they are used together with `\AssignTemplateKeys` then the latter command should come first in the template code.

The main use case for these commands is the situation where there is an argument (normally #1) to the template in which a key/value list can be specified that overwrites the normal settings. In that case one could use

`\SetKnownTemplateKeys{<type>}{<template>}{#1}`

to process this key/value list inside the template.

If `\SetKnownTemplateKeys` is executed and the `<keyvals>` argument contains keys not known to the `<template>` they are simply ignored and stored in the tokenlist `\UnusedTemplateKeys` without generating an error. This way it is possible to apply the same key/val list specified by the user on a document-level command or environment to several templates, which is useful, if the command or environment is implemented by calling several different template instances.

As a variation of that, you can use this key/val list the first time, and for the next template instance use what remains in `\UnusedTemplateKeys` (i.e., the key/val list with only the keys that have not been processed previously). The final processing step could then be `\SetTemplateKeys`, which unconditionally attempts to set the `<keyvals>` received in its third argument. This command complains if any of them are unknown keys. Alternatively, you could use `\SetKnownTemplateKeys` and afterwards check whether `\UnusedTemplateKeys` is empty.¹

For example, a list, such as `enumerate`, is made up from a `blockenv`, `block`, `list`, and a `para` template and in the single user-supplied optional argument of `enumerate` key/values for any of these templates might be specified.

In fact, in the particular example of list environments, the supplied key/value list is also saved and then applied to each `\item` which is implemented through an `item` template. This way, one can specify one-off settings for all the items of a single list (on the environment level), as well as to individual items within that list (by specifying them in the optional argument of an `\item`). With `\SetKnownTemplateKeys` and `\SetTemplateKeys` working together, it is possible to provide this flexibility and still alert the user when one of their keys is misspelled.

On the other hand you may want to allow for “misspellings” without generating an error or a warning. For example, if you define a template that accepts only a few keys, you might just want to ignore anything specified in the source when you use this template in place of a different one, without the need to alter the document source. Or you might

¹Using `\SetTemplateKeys` exposes the inner structure of the template keys when generating an error. This is something one may want to avoid as it can be confusing to the user, especially if several templates are involved. In that case use `\SetKnownTemplateKeys` and afterwards check whether `\UnusedTemplateKeys` is empty; if it is not empty then generate your own error message.

just generate a warning message, which is easy, given that the unused key/values are available in the `\UnusedTemplateKeys` variable.

```
\DeclareTemplateCopy \DeclareTemplateCopy
    {\type} {\template2} {\template1}
```

Copies `\template1` of `\type` to a new name `\template2`: the copy can then be edited independent of the original.

6 Multiple choices

The `choice` key type implements multiple choice input. At the interface level, only the list of valid choices is needed:

```
\DeclareTemplateInterface { foo } { bar } { 0 }
    { key-name : choice { A, B, C } }
```

where the choices are given as a comma-list (which must therefore be wrapped in braces). A default value can also be given:

```
\DeclareTemplateInterface { foo } { bar } { 0 }
    { key-name : choice { A, B, C } = A }
```

At the implementation level, each choice is associated with code, using a nested key-value list.

```
\DeclareTemplateCode { foo } { bar } { 0 }
{
    key-name =
    {
        A = Code-A ,
        B = Code-B ,
        C = Code-C
    }
}
{ ... }
```

The two choice lists should match, but in the implementation a special `unknown` choice is also available. This can be used to ignore values and implement an “else” branch:

```
\DeclareTemplateCode { foo } { bar } { 0 }
{
    key-name =
    {
        A      = Code-A ,
        B      = Code-B ,
        C      = Code-C ,
        unknown = Else-code
    }
}
{ ... }
```

The **unknown** entry must be the last one given, and should *not* be listed in the interface part of the template.

For keys which accept the values **true** and **false** both the boolean and choice key types can be used. As template interfaces are intended to prompt clarity at the design level, the boolean key type should be favored, with the choice type reserved for keys which take arbitrary values.

7 Instances

After a template is defined it still needs to be put to use. The parameters that it expects need to be defined before it can be used in a document. Every time a template has parameters given to it, an *instance* is created, and this is the code that ends up in the document to perform the typesetting of whatever pieces of information are input into it.

For example, a template might say “here is a section with or without a number that might be centered or left aligned and print its contents in a certain font of a certain size, with a bit of a gap before and after it” whereas an instance declares “this is a section with a number, which is centered and set in 12pt italic with a 10pt skip before and a 12pt skip after it”. Therefore, an instance is just a frozen version of a template with specific settings as chosen by the designer.

<code>\DeclareInstance</code>	<code>\DeclareInstance</code> <code>{\langle type \rangle} {\langle instance \rangle} {\langle template \rangle} {\langle parameters \rangle}</code>
-------------------------------	---

This function uses a `\langle template \rangle` for an `\langle type \rangle` to create an `\langle instance \rangle`. The `\langle instance \rangle` will be set up using the `\langle parameters \rangle`, which will set some of the `\langle keys \rangle` in the `\langle template \rangle`.

As a practical example, consider a type for document sections (which might include chapters, parts, sections, *etc.*), which is called **sectioning**. One possible template for this type might be called **basic**, and one instance of this template would be a numbered section. The instance declaration might read:

```
\DeclareInstance { sectioning } { section-num } { basic }
{
  numbered      = true ,
  justification = center ,
  font          = \normalsize\itshape ,
  before-skip   = 10pt ,
  after-skip    = 12pt ,
}
```

Of course, the key names here are entirely imaginary, but illustrate the general idea of fixing some settings.

<code>\IfInstanceExistsT</code>	<code>\IfInstanceExistsTF {\langle type \rangle} {\langle instance \rangle} {\langle true code \rangle} {\langle false code \rangle}</code>
<code>\IfInstanceExistsF</code>	
<code>\IfInstanceExistsTF</code>	Tests if the named <code>\langle instance \rangle</code> of a <code>\langle type \rangle</code> exists, and then inserts the appropriate code into the input stream.

<code>\DeclareInstanceCopy</code>	<code>\DeclareInstanceCopy</code> <code>{\langle type \rangle} {\langle instance2 \rangle} {\langle instance1 \rangle}</code>
-----------------------------------	--

Copies the `\langle values \rangle` for `\langle instance1 \rangle` for an `\langle type \rangle` to `\langle instance2 \rangle`.

8 Document interface

After the instances have been chosen, document commands must be declared to use those instances in the document. `\UseInstance` calls instances directly, and this command should be used internally in document-level mark-up.

<hr/> <hr/>	<code>\UseInstance</code>
	<code>{\type}\{instance\} \langle arguments \rangle</code>
	Uses an $\langle instance \rangle$ of the $\langle type \rangle$, which will require $\langle arguments \rangle$ as determined by the number specified for the $\langle type \rangle$. The $\langle instance \rangle$ must have been declared before it can be used, otherwise an error is raised.

<hr/> <hr/>	<code>\UseTemplate</code>
	<code>\UseTemplate {\type}\{template\}</code> <code>\{settings\} \langle arguments \rangle</code>
	Uses the $\langle template \rangle$ of the specified $\langle type \rangle$, applying the $\langle settings \rangle$ and absorbing $\langle arguments \rangle$ as detailed by the $\langle type \rangle$ declaration. This in effect is the same as creating an instance using <code>\DeclareInstance</code> and immediately using it with <code>\UseInstance</code> , but without the instance having any further existence. This command is therefore useful when a template needs to be used only once.

This function can also be used as the argument to `instance` key types:

```
\DeclareInstance { type } { template } { instance }
{
  instance-key =
    \UseTemplate { type2 } { template2 } { <settings> }
}
```

9 Changing existing definitions

Template parameters may be assigned specific defaults for instances to use if the instance declaration doesn't explicit set those parameters. In some cases, the document designer will wish to edit these defaults to allow them to “cascade” to the instances. The alternative would be to set each parameter identically for each instance declaration, a tedious and error-prone process.

<hr/> <hr/>	<code>\EditTemplateDefaults</code>
	<code>\EditTemplateDefaults</code> <code>{\type}\{template\} {\new defaults}</code>
	Edits the $\langle defaults \rangle$ for a $\langle template \rangle$ for an $\langle type \rangle$. The $\langle new defaults \rangle$, given as a key–value list, replace the existing defaults for the $\langle template \rangle$. This means that the change will apply to instances declared after the editing, but that instances which have already been created are unaffected.

<hr/> <hr/>	<code>\EditInstance</code>
	<code>{\type}\{instance\} {\new values}</code>
	Edits the $\langle values \rangle$ for an $\langle instance \rangle$ for an $\langle type \rangle$. The $\langle new values \rangle$, given as a key–value list, replace the existing values for the $\langle instance \rangle$. This function is complementary to <code>\EditTemplateDefaults</code> : <code>\EditInstance</code> changes a single instance while leaving the template untouched.

10 Getting information about templates and instances

<hr/> <hr/>	<code>\ShowInstanceValues {<type>} {<instance>}</code>	Shows the <i><values></i> for an <i><instance></i> of the given <i><type></i> at the terminal.
<hr/> <hr/>	<code>\ShowTemplateCode {<type>} {<template>}</code>	Shows the <i><code></i> of a <i><template></i> for an <i><type></i> in the terminal.
<hr/> <hr/>	<code>\ShowTemplateDefaults {<type>} {<template>}</code>	Shows the <i><default></i> values of a <i><template></i> for an <i><type></i> in the terminal.
<hr/> <hr/>	<code>\ShowTemplateInterface {<type>} {<template>}</code>	Shows the <i><keys></i> and associated <i><key types></i> of a <i><template></i> for an <i><type></i> in the terminal.
<hr/> <hr/>	<code>\ShowTemplateVariables {<type>} {<template>}</code>	Shows the <i><variables></i> and associated <i><keys></i> of a <i><template></i> for an <i><type></i> in the terminal. Note that <i>code</i> and <i>choice</i> keys do not map directly to variables but to arbitrary code. For <i>choice</i> keys, each valid choice is shown as a separate entry in the list, with the key name and choice separated by a space, for example

Template 'example' of type 'example' has variable mapping:

```
> demo unknown => \def \demo {?}
> demo c      => \def \demo {c}
> demo b      => \def \demo {b}
> demo a      => \def \demo {a}.
```

would be shown for a choice key *demo* with valid choices *a*, *b* and *c*, plus code for an *unknown* branch.

11 The implementation

```
1 <@@=template>
2 <*2ekernel>
3 \message{templates,}
4 </2ekernel>
5 <*2ekernel | latexrelease>
6 \ExplSyntaxOn
7 <latexrelease> \NewModuleRelease{2024/06/01}{ltemplates}
8 <latexrelease> {Prototype~document~commands}%
```

11.1 Variables and constants

```

\c__template_code_root_tl
\c__template_defaults_root_tl
\c__template_instances_root_tl
\c__template_keytypes_root_tl
\c__template_key_order_root_tl
\c__template_restrict_root_tl
\c__template_values_root_tl
\c__template_vars_root_tl

```

So that literal values are kept to a minimum.

```

 9 \tl_const:Nn \c__template_code_root_tl      { template~code~>~ }
10 \tl_const:Nn \c__template_defaults_root_tl  { template~defaults~>~ }
11 \tl_const:Nn \c__template_instances_root_tl { template~instance~>~ }
12 \tl_const:Nn \c__template_keytypes_root_tl  { template~key~types~>~ }
13 \tl_const:Nn \c__template_key_order_root_tl { template~key~order~>~ }
14 \tl_const:Nn \c__template_values_root_tl     { template~values~>~ }
15 \tl_const:Nn \c__template_vars_root_tl      { template~vars~>~ }

```

```

\c__template_keytypes_arg_seq

```

A list of keytypes which also need additional data (an argument), used to parse the keytype correctly.

```

16 \seq_const_from_clist:Nn \c__template_keytypes_arg_seq
17   { choice , function , instance }

```

```

\g__template_type_prop

```

For storing types and the associated number of arguments.

```

18 \prop_new:N \g__template_type_prop

```

```

\l__template_assignments_tl

```

When creating an instance, the assigned values are collected here.

```

19 \tl_new:N \l__template_assignments_tl

```

```

\l__template_default_tl

```

The default value for a key is recovered here from the property list in which it is stored.

```

20 \tl_new:N \l__template_default_tl

```

```

\l__template_error_bool

```

A flag for errors to be carried forward.

```

21 \bool_new:N \l__template_error_bool

```

```

\l__template_global_bool

```

Used to indicate that assignments should be global.

```

22 \bool_new:N \l__template_global_bool

```

```

\l__template_key_name_tl
\l__template_keytype_tl
\l__template_keytype_arg_tl
\l__template_value_tl
\l__template_var_tl

```

When defining each key in a template, the name and type of the key need to be separated and stored. Any argument needed by the keytype is also stored separately.

```

23 \tl_new:N \l__template_key_name_tl
24 \tl_new:N \l__template_keytype_tl
25 \tl_new:N \l__template_keytype_arg_tl
26 \tl_new:N \l__template_value_tl
27 \tl_new:N \l__template_var_tl

```

```

\l__template_keytypes_prop
\l__template_key_order_seq
\l__template_values_prop
\l__template_vars_prop

```

To avoid needing too many difficult-to-follow csname assignments, various scratch token registers are used to build up data, which is then transferred

```

28 \prop_new:N \l__template_keytypes_prop
29 \seq_new:N \l__template_key_order_seq
30 \prop_new:N \l__template_values_prop
31 \prop_new:N \l__template_vars_prop

```

```

\l__template_tmp_clist
\l__template_tmp_dim
\l__template_tmp_int
\l__template_tmp_muskip
\l__template_tmp_skip
\l__template_tmp_tl

```

Scratch space.

```

32 \clist_new:N \l__template_tmp_clist
33 \dim_new:N \l__template_tmp_dim
34 \int_new:N \l__template_tmp_int
35 \muskip_new:N \l__template_tmp_muskip
36 \skip_new:N \l__template_tmp_skip
37 \tl_new:N \l__template_tmp_tl

```

```

\s__template_mark
\s__template_stop

```

Internal scan marks.

```

38 \scan_new:N \s__template_mark
39 \scan_new:N \s__template_stop

```

```

\q__template_nil

```

Internal quarks.

```

40 \quark_new:N \q__template_nil

```

```

\__template_quark_if_nil_p:n
\__template_quark_if_nil:nTF

```

Branching quark conditional.

```

41 \__kernel_quark_new_conditional:Nn \__template_quark_if_nil:N { F }

```

(End of definition for __template_quark_if_nil:nTF.)

11.2 Testing existence and validity

There are a number of checks needed for either the existence of a type, template or instance. There are also some for the validity of a particular call. All of these are collected up here.

`__template_execute_if_arg_agree:nnT`

A test agreement between the number of arguments for the template type and that specified when creating a template. This is not done as a separate conditional for efficiency and better error message

```

42 \cs_new_protected:Npn \__template_execute_if_arg_agree:nnT #1#2#3
43 {
44   \prop_get:NnN \g__template_type_prop {#1} \l__template_tmp_tl
45   \int_compare:nNnTF {#2} = \l__template_tmp_tl
46     {#3}
47   {
48     \msg_error:nneee { template } { argument-number-mismatch }
49     {#1} { \l__template_tmp_tl } {#2}
50   }
51 }
```

(End of definition for __template_execute_if_arg_agree:nnT.)

`__template_execute_if_code_exist:nnT`

A template is only fully declared if the code has been set up, which can be checked by looking for the template function itself.

```

52 \cs_new_protected:Npn \__template_execute_if_code_exist:nnT #1#2#3
53 {
54   \cs_if_exist:cTF { \c__template_code_root_tl #1 / #2 }
55     {#3}
56     { \msg_error:nnnn { template } { no-template-code } {#1} {#2} }
57 }
```

(End of definition for __template_execute_if_code_exist:nnT.)

`__template_execute_if_keytype_exist:nT`

`__template_execute_if_keytype_exist:VT`

The test for valid keytypes looks for a function to set up the key, which is part of the “code” side of the template definition. This avoids having different lists for the two parts of the process.

```

58 \cs_new_protected:Npn \__template_execute_if_keytype_exist:nT #1#2
59 {
60   \cs_if_exist:cTF { __template_store_value_ #1 :n }
61     {#2}
62     { \msg_error:nnn { template } { unknown-keytype } {#1} }
63 }
64 \cs_generate_variant:Nn \__template_execute_if_keytype_exist:nT { V }
```

(End of definition for __template_execute_if_keytype_exist:nT.)

`__template_execute_if_type_exist:nT`

To check that a particular type is valid.

```

65 \cs_new_protected:Npn \__template_execute_if_type_exist:nT #1#2
66 {
67   \prop_if_in:NnTF \g__template_type_prop {#1}
68     {#2}
69     { \msg_error:nnn { template } { unknown-type } {#1} }
70 }
```

(End of definition for __template_execute_if_type_exist:nT.)

`_template_execute_if_keys_exist:nnT` To check that the keys for a template have been set up before trying to create any code, a simple check for the correctly-named keytype property list.

```

71 \cs_new_protected:Npn \_template\_if\_keys\_exist:nnT #1#2#3
72 {
73   \cs\_if\_exist:cTF { \c\_template\_keytypes\_root\_tl #1 / #2 }
74   {#3}
75   { \msg\_error:nnnn { template } { unknown-template } {#1} {#2} }
76 }

```

(End of definition for `_template_execute_if_keys_exist:nnT`.)

`_template_if_key_value:nTF` Tests for the first token in a string being `\KeyValue`.

`_template_if_key_value:VTF`

```

77 \prg\_new\_conditional:Npnn \_template\_if\_key\_value:n #1 { T , F , TF }
78 {
79   \str\_if\_eq:noTF { \KeyValue } { \tl\_head:w #1 \q\_nil \q\_stop }
80   \prg\_return\_true:
81   \prg\_return\_false:
82 }
83 \prg\_generate\_conditional\_variant:Nnn \_template\_if\_key\_value:n { V } { T , F , TF }

```

(End of definition for `_template_if_key_value:nTF`.)

`_template_if_instance_exist:nnTF` Testing for an instance

```

84 \prg\_new\_conditional:Npnn \_template\_if\_instance\_exist:nn #1#2 { T, F, TF }
85 {
86   \cs\_if\_exist:cTF { \c\_template\_instances\_root\_tl #1 / #2 }
87   \prg\_return\_true:
88   \prg\_return\_false:
89 }

```

(End of definition for `_template_if_instance_exist:nnTF`.)

`_template_if_use_template:nTF` Tests for the first token in a string being `\UseTemplate`.

```

90 \prg\_new\_conditional:Npnn \_template\_if\_use\_template:n #1 { TF }
91 {
92   \str\_if\_eq:noTF { \UseTemplate } { \tl\_head:w #1 \q\_nil \q\_stop }
93   \prg\_return\_true:
94   \prg\_return\_false:
95 }

```

(End of definition for `_template_if_use_template:nTF`.)

11.3 Saving and recovering property lists

The various property lists for templates have to be shuffled in and out of storage.

`_template_store_defaults:nn`

`_template_store_keytypes:nn`

The defaults and keytypes are transferred from the scratch property lists to the “proper” lists for the template being created.

`_template_store_values:nn`

`_template_store_vars:nn`

```

96 \cs\_new\_protected:Npn \_template\_store\_defaults:nn #1#2
97 {
98   \debug\_suspend:
99   \prop\_gclear\_new:c { \c\_template\_defaults\_root\_tl #1 / #2 }
100   \prop\_gset\_eq:cN { \c\_template\_defaults\_root\_tl #1 / #2 }
101   \l\_template\_values\_prop

```

```

102     \debug_resume:
103 }
104 \cs_new_protected:Npn \__template_store_keytypes:nn #1#2
105 {
106     \debug_suspend:
107     \prop_if_exist:cTF { \c__template_keytypes_root_tl #1 / #2 }
108     {
109         \msg_info:nnnn { template } { declare-template-interface } {#1} {#2}
110         \prop_gclear:c { \c__template_keytypes_root_tl #1 / #2 }
111     }
112     { \prop_new:c { \c__template_keytypes_root_tl #1 / #2 } }
113     \prop_gset_eq:cN { \c__template_keytypes_root_tl #1 / #2 }
114     \l__template_keytypes_prop
115     \seq_gclear_new:c { \c__template_key_order_root_tl #1 / #2 }
116     \seq_gset_eq:cN { \c__template_key_order_root_tl #1 / #2 }
117     \l__template_key_order_seq
118     \debug_resume:
119 }
120 \cs_new_protected:Npn \__template_store_values:nn #1#2
121 {
122     \debug_suspend:
123     \prop_clear_new:c { \c__template_values_root_tl #1 / #2 }
124     \prop_set_eq:cN { \c__template_values_root_tl #1 / #2 }
125     \l__template_values_prop
126     \debug_resume:
127 }
128 \cs_new_protected:Npn \__template_store_vars:nn #1#2
129 {
130     \debug_suspend:
131     \prop_gclear_new:c { \c__template_vars_root_tl #1 / #2 }
132     \prop_gset_eq:cN { \c__template_vars_root_tl #1 / #2 }
133     \l__template_vars_prop
134     \debug_resume:
135 }

```

(End of definition for __template_store_defaults:nn and others.)

__template_recover_defaults:nn Recovering the stored data for a template is rather less complex than storing it. All that happens is the data is transferred from the permanent to the scratch storage. However, we need to check the scratch storage does exist.

__template_recover_keytypes:nn

__template_recover_values:nn

__template_recover_vars:nn

```

136 \cs_new_protected:Npn \__template_recover_defaults:nn #1#2
137 {
138     \prop_if_exist:cTF
139     { \c__template_defaults_root_tl #1 / #2 }
140     {
141         \prop_set_eq:Nc \l__template_values_prop
142         { \c__template_defaults_root_tl #1 / #2 }
143     }
144     { \prop_clear:N \l__template_values_prop }
145 }
146 \cs_new_protected:Npn \__template_recover_keytypes:nn #1#2
147 {
148     \prop_if_exist:cTF
149     { \c__template_keytypes_root_tl #1 / #2 }

```

```

150     {
151         \prop_set_eq:Nc \l__template_keytypes_prop
152         { \c__template_keytypes_root_tl #1 / #2 }
153     }
154     { \prop_clear:N \l__template_keytypes_prop }
155 \seq_if_exist:cTF { \c__template_key_order_root_tl #1 / #2 }
156     {
157         \seq_set_eq:Nc \l__template_key_order_seq
158         { \c__template_key_order_root_tl #1 / #2 }
159     }
160     { \seq_clear:N \l__template_key_order_seq }
161 }
162 \cs_new_protected:Npn \__template_recover_values:nn #1#2
163 {
164     \prop_if_exist:cTF
165     { \c__template_values_root_tl #1 / #2 }
166     {
167         \prop_set_eq:Nc \l__template_values_prop
168         { \c__template_values_root_tl #1 / #2 }
169     }
170     { \prop_clear:N \l__template_values_prop }
171 }
172 \cs_new_protected:Npn \__template_recover_vars:nn #1#2
173 {
174     \prop_if_exist:cTF
175     { \c__template_vars_root_tl #1 / #2 }
176     {
177         \prop_set_eq:Nc \l__template_vars_prop
178         { \c__template_vars_root_tl #1 / #2 }
179     }
180     { \prop_clear:N \l__template_vars_prop }
181 }

```

(End of definition for `__template_recover_defaults:nn` and others.)

11.4 Creating new template types

`__template_define_type:nn` Although the type is the “top level” of the template system, it is actually very easy to implement. All that happens is that the number of arguments required is recorded, indexed by the name of the type.

```

182 \cs_new_protected:Npn \__template_define_type:nn #1#2
183 {
184     \prop_if_in:NnTF \g__template_type_prop {#1}
185     { \msg_error:nnn { template } { type-already-defined } {#1} }
186     { \__template_declare_type:nn {#1} {#2} }
187 }
188 \cs_new_protected:Npn \__template_declare_type:nn #1#2
189 {
190     \int_set:Nn \l__template_tmp_int {#2}
191     \int_compare:nTF { 0 <= \l__template_tmp_int <= 9 }
192     {
193         \msg_info:nnnV { template } { declare-type }
194         {#1} \l__template_tmp_int
195         \prop_gput:NnV \g__template_type_prop {#1}

```



```

196         \l__template_tmp_int
197     }
198     {
199         \msg_error:nnnV { template } { bad-number-of-arguments }
200         {#1} \l__template_tmp_int
201     }
202 }

```

(End of definition for __template_define_type:nn and __template_declare_type:nn.)

11.5 Design part of template declaration

The “design” part of a template declaration defines the general behaviour of each key, and possibly a default value. However, it does not include the implementation. This means that what happens here is the two properties are saved to appropriate lists, which can then be used later to recover the information when implementing the keys.

__template_declare_template_keys:nnnn

The main function for the “design” part of creating a template starts by checking that the type exists and that the number of arguments required agree. If that is all fine, then the two storage areas for defaults and keytypes are initialised. The mechanism is then set up for the l3keys module to actually parse the keys. Finally, the code hands of to the storage routine to save the parsed information properly.

```

203 \cs_new_protected:Npn \__template_declare_template_keys:nnnn #1#2#3#4
204 {
205     \__template_execute_if_type_exist:nT {#1}
206     {
207         \__template_execute_if_arg_agree:nnT {#1} {#3}
208         {
209             \prop_clear:N \l__template_values_prop
210             \prop_clear:N \l__template_keytypes_prop
211             \seq_clear:N \l__template_key_order_seq
212             \keyval_parse:NNn
213             \__template_parse_keys_elt:n \__template_parse_keys_elt:nn {#4}
214             \__template_store_defaults:nn {#1} {#2}
215             \__template_store_keytypes:nn {#1} {#2}
216         }
217     }
218 }

```

(End of definition for __template_declare_template_keys:nnnn.)

__template_parse_keys_elt:n

__template_parse_keys_elt_aux:n

__template_parse_keys_elt_aux:

Processing the key part of the key–value pair is always carried out using this function, even if a value was found. First, the key name is separated from the keytype, and if necessary the keytype is separated into two parts. This information is then used to check that the keytype is valid, before storing the keytype (plus argument if necessary) as a property of the key name. The key name is also stored (in braces) in the token list to record the order the keys are defined in.

```

219 \cs_new_protected:Npn \__template_parse_keys_elt:n #1
220 {
221     \__template_split_keytype:n {#1}
222     \bool_if:NF \l__template_error_bool
223     {
224         \__template_execute_if_keytype_exist:VT \l__template_keytype_t1

```

```

225     {
226         \seq_map_function:NN \c__template_keytypes_arg_seq
227         \__template_parse_keys_elt_aux:n
228         \bool_if:NF \l__template_error_bool
229         {
230             \seq_if_in:NoTF \l__template_key_order_seq
231             \l__template_key_name_tl
232             {
233                 \msg_error:nnV { template } { duplicate-key-interface }
234                 \l__template_key_name_tl
235             }
236             { \__template_parse_keys_elt_aux: }
237         }
238     }
239 }
240 }
241 \cs_new_protected:Npn \__template_parse_keys_elt_aux:n #1
242 {
243     \str_if_eq:VnT \l__template_keytype_tl {#1}
244     {
245         \tl_if_empty:NT \l__template_keytype_arg_tl
246         {
247             \msg_error:nnn { template } { keytype-requires-argument } {#1}
248             \bool_set_true:N \l__template_error_bool
249             \seq_map_break:
250         }
251     }
252 }
253 \cs_new_protected:Npn \__template_parse_keys_elt_aux:
254 {
255     \tl_set:Ne \l__template_tmp_tl
256     {
257         \l__template_keytype_tl
258         \tl_if_empty:NF \l__template_keytype_arg_tl
259         { { \l__template_keytype_arg_tl } }
260     }
261     \prop_put:NVV \l__template_keytypes_prop \l__template_key_name_tl
262     \l__template_tmp_tl
263     \seq_put_right:NV \l__template_key_order_seq \l__template_key_name_tl
264     \str_if_eq:VnT \l__template_keytype_tl { choice }
265     {
266         \clist_if_in:NnT \l__template_keytype_arg_tl { unknown }
267         { \msg_error:nn { template } { choice-unknown-reserved } }
268     }
269 }

```

(End of definition for __template_parse_keys_elt:n, __template_parse_keys_elt_aux:n, and __template_parse_keys_elt_aux:.)

__template_parse_keys_elt:nn For keys which have a default, the keytype and key name are first separated out by the __template_parse_keys_elt:n routine, before storing the default value in the scratch property list.

```

270 \cs_new_protected:Npn \__template_parse_keys_elt:nn #1#2
271 {

```

```

272     \__template_parse_keys_elt:n {#1}
273     \use:c { __template_store_value_ \l__template_keytype_tl :n } {#2}
274 }

```

(End of definition for __template_parse_keys_elt:nn.)

```

\__template_split_keytype:n
  \__template_split_keytype_aux:w

```

The keytype and key name should be separated by :. As the definition might be given inside or outside of a code block, the category code of colons is standardised. After that, the standard delimited argument method is used to separate the two parts.

```

275 \cs_new_protected:Npe \__template_split_keytype:n #1
276 {
277   \exp_not:N \bool_set_false:N \exp_not:N \l__template_error_bool
278   \tl_set:Nn \exp_not:N \l__template_tmp_tl {#1}
279   \tl_replace_all:Nnn \exp_not:N \l__template_tmp_tl { : } { \token_to_str:N : }
280   \tl_if_in:NnTF \exp_not:N \l__template_tmp_tl { \token_to_str:N : }
281   {
282     \exp_not:n
283     {
284       \tl_clear:N \l__template_key_name_tl
285       \exp_after:wN \__template_split_keytype_aux:w
286       \l__template_tmp_tl \s__template_stop
287     }
288   }
289   {
290     \exp_not:N \bool_set_true:N \exp_not:N \l__template_error_bool
291     \msg_error:nnn { template } { missing-keytype } {#1}
292   }
293 }
294 \use:e
295 {
296   \cs_new_protected:Npn \exp_not:N \__template_split_keytype_aux:w
297   #1 \token_to_str:N : #2 \s__template_stop
298   {
299     \tl_put_right:Ne \exp_not:N \l__template_key_name_tl
300     {
301       \exp_not:N \tl_trim_spaces:e
302       { \exp_not:N \tl_to_str:n {#1} }
303     }
304     \tl_if_in:nnTF {#2} { \token_to_str:N : }
305     {
306       \tl_put_right:Nn \exp_not:N \l__template_key_name_tl
307       { \token_to_str:N : }
308       \exp_not:N \__template_split_keytype_aux:w #2 \s__template_stop
309     }
310     {
311       \exp_not:N \tl_if_empty:NTF \exp_not:N \l__template_key_name_tl
312       {
313         \msg_error:nnn { template } { empty-key-name }
314         { \token_to_str:N : #2 }
315       }
316       { \exp_not:N \__template_split_keytype_arg:n {#2} }
317     }
318   }
319 }

```

(End of definition for `__template_split_keytype:n` and `__template_split_keytype_aux:w`.)

`__template_split_keytype_arg:n`
`__template_split_keytype_arg:V`
`__template_split_keytype_arg_aux:n`
`__template_split_keytype_arg_aux:w`

The second stage of sorting out the keytype is to check for an argument. As there is no convenient delimiting token to look for, a check is made instead for each possible text value for the keytype. To keep things faster, this only involves the keytypes that need an argument. If a match is made, then a check is also needed to see that it is at the start of the keytype information. All being well, the split can then be applied. Any non-matching keytypes are assumed to be “correct” as given, and are left alone (this is checked by other code).

```

320 \cs_new_protected:Npn __template_split_keytype_arg:n #1
321 {
322   \tl_set:Nc \l__template_keytype_tl { \tl_trim_spaces:n {#1} }
323   \tl_clear:N \l__template_keytype_arg_tl
324   \cs_set_protected:Npn __template_split_keytype_arg_aux:n ##1
325   {
326     \tl_if_in:nnT {#1} {##1}
327     {
328       \cs_set:Npn __template_split_keytype_arg_aux:w
329       #####1 ##1 #####2 \s__template_stop
330       {
331         \tl_if_blank:nT {#####1}
332         {
333           \tl_set:Nc \l__template_keytype_tl
334           { \tl_trim_spaces:n {##1} }
335           \tl_if_blank:nF {#####2}
336           {
337             \tl_set:Nc \l__template_keytype_arg_tl
338             { \use:n #####2 }
339           }
340           \seq_map_break:
341         }
342       }
343       __template_split_keytype_arg_aux:w #1 \s__template_stop
344     }
345   }
346   \seq_map_function:NN \c__template_keytypes_arg_seq
347   __template_split_keytype_arg_aux:n
348 }
349 \cs_generate_variant:Nn __template_split_keytype_arg:n { V }
350 \cs_new:Npn __template_split_keytype_arg_aux:n #1 { }
351 \cs_new:Npn __template_split_keytype_arg_aux:w #1 \s__template_stop { }

```

(End of definition for `__template_split_keytype_arg:n`, `__template_split_keytype_arg_aux:n`, and `__template_split_keytype_arg_aux:w`.)

11.5.1 Storing values

As `ltemplates` pre-processes key values for efficiency reasons, there is a need to convert the values given as defaults into “ready to use” data. The same general idea is true when an instance is declared. However, assignments are not made until an instance is used, and so there has to be some intermediate storage. Furthermore, the ability to delay evaluation of results is needed. To achieve these aims, a series of “process and store” functions are defined here.

All of the information about the key (the key name and the keytype) is already stored as variables. The same property list is always used to store the data, meaning that the only argument required is the value to be processed and potentially stored.

`_template_store_value_boolean:n`

```
352 \cs_new_protected:Npn \_template_store_value_boolean:n #1
353 { \prop_put:Non \l__template_values_prop \l__template_key_name_tl {#1} }
```

(End of definition for `_template_store_value_boolean:n`.)

`__template_store_value:n`

With no need to worry about delayed evaluation, these keytypes all just store the input directly.

`_template_store_value_choice:n`

`_template_store_value_function:n`

`_template_store_value_instance:n`

```
354 \cs_new_protected:Npn \__template_store_value:n #1
355 { \prop_put:Non \l__template_values_prop \l__template_key_name_tl {#1} }
356 \cs_new_eq:NN \_template_store_value_choice:n \__template_store_value:n
357 \cs_new_eq:NN \_template_store_value_function:n \__template_store_value:n
358 \cs_new_eq:NN \_template_store_value_instance:n \__template_store_value:n
```

(End of definition for `__template_store_value:n` and others.)

`_template_store_value_aux:Nn`

Storing values in `\l__template_values_prop` is in most cases the same.

`_template_store_value_integer:n`

`_template_store_value_length:n`

`_template_store_value_muskip:n`

`_template_store_value_real:n`

`_template_store_value_skip:n`

`_template_store_value_tokenlist:n`

`_template_store_value_commalist:n`

```
359 \cs_new_protected:Npn \_template_store_value_aux:Nn #1#2
360 { \prop_put:Non \l__template_values_prop \l__template_key_name_tl {#2} }
361 \cs_new_protected:Npn \_template_store_value_integer:n
362 { \_template_store_value_aux:Nn \int_eval:n }
363 \cs_new_protected:Npn \_template_store_value_length:n
364 { \_template_store_value_aux:Nn \dim_eval:n }
365 \cs_new_protected:Npn \_template_store_value_muskip:n
366 { \_template_store_value_aux:Nn \muskip_eval:n }
367 \cs_new_protected:Npn \_template_store_value_real:n
368 { \_template_store_value_aux:Nn \fp_eval:n }
369 \cs_new_protected:Npn \_template_store_value_skip:n
370 { \_template_store_value_aux:Nn \skip_eval:n }
371 \cs_new_protected:Npn \_template_store_value_tokenlist:n
372 { \_template_store_value_aux:Nn \use:n }
373 \cs_new_eq:NN \_template_store_value_commalist:n \_template_store_value_tokenlist:n
```

(End of definition for `_template_store_value_aux:Nn` and others.)

11.6 Implementation part of template declaration

`_template_declare_template_code:nnnnn`

The main function for implementing a template starts with a couple of simple checks to make sure that there are no obvious mistakes: the number of arguments must agree and the template keys must have been declared.

`_template_declare_template_code:nnnn`

```
374 \cs_new_protected:Npn \_template_declare_template_code:nnnnn #1#2#3#4#5
375 {
376   \_template_execute_if_type_exist:nT {#1}
377   {
378     \_template_execute_if_arg_agree:nnT {#1} {#3}
379     {
380       \_template_if_keys_exist:nnT {#1} {#2}
381       {
382         \_template_store_key_implementation:nnn {#1} {#2} {#4}
383         \str_if_in:nnTF {#5} { AssignTemplateKeys }
```

```

384         {
385             \regex_match:nnTF { \c { AssignTemplateKeys } } {#5}
386             { \_template_declare_template_code:nnnn {#1} {#2} {#3} {#5} }
387             {
388                 \_template_declare_template_code:nnnn
389                 {#1} {#2} {#3} { \AssignTemplateKeys #5 }
390             }
391         }
392     {
393         \_template_declare_template_code:nnnn
394         {#1} {#2} {#3} { \AssignTemplateKeys #5 }
395     }
396 }
397 }
398 }
399 }
400 \cs_new_protected:Npn \_template_declare_template_code:nnnn #1#2#3#4
401 {
402     \cs_if_exist:cT { \c__template_code_root_tl #1 / #2 }
403     { \msg_info:nnnn { template } { declare-template-code } {#1} {#2} }
404     \cs_generate_from_arg_count:cNnn
405     { \c__template_code_root_tl #1 / #2 }
406     \cs_gset_protected:Npn {#3} {#4}
407 }

```

(End of definition for `_template_declare_template_code:nnnn` and `_template_declare_template_code:nnnn`.)

`_template_store_key_implementation:nnn`

Actually storing the implementation part of a template is quite easy as it only requires the list of keys given to be turned into a property list. There is also some error-checking to do, hence the need to have the list of defined keytypes available. In certain cases (when choices are involved) parsing the key results in changes to the default values. That is why they are loaded and then saved again.

```

408 \cs_new_protected:Npn \_template_store_key_implementation:nnn #1#2#3
409 {
410     \_template_recover_defaults:nn {#1} {#2}
411     \_template_recover_keytypes:nn {#1} {#2}
412     \prop_clear:N \l__template_vars_prop
413     \keyval_parse:nnn
414     { \_template_parse_vars_elt:n } { \_template_parse_vars_elt:nnn { #1 / #2 } } {#3}
415     \_template_store_vars:nn {#1} {#2}
416     \prop_map_inline:Nn \l__template_keytypes_prop
417     { \msg_error:nnnnn { template } { key-not-implemented } {##1} {#2} {#1} }
418 }

```

(End of definition for `_template_store_key_implementation:nnn`.)

`_template_parse_vars_elt:n`

At the implementation stage, every key must have a value given. So this is an error function.

```

419 \cs_new_protected:Npn \_template_parse_vars_elt:n #1
420 { \msg_error:nnn { template } { key-no-variable } {#1} }

```

(End of definition for `_template_parse_vars_elt:n`.)

```

    \_template_parse_vars_elt:nnn
    \_template_parse_vars_elt_aux:nn
    \_template_parse_vars_elt_aux:nw
    \_template_parse_vars_elt_aux:nnn
    \_template_parse_vars_elt_aux:nne
    \_template_parse_vars_elt_key:nn

```

The actual storage part here is very simple: the storage bin name is placed into the property list. At the same time, a comparison is made with the keytypes defined earlier: if there is a mismatch then an error is raised.

```

421 \cs_new_protected:Npn \_template_parse_vars_elt:nnn #1#2#3
422 {
423   \tl_set:Nx \l__template_key_name_tl
424     { \tl_trim_spaces:e { \tl_to_str:n {#2} } }
425   \prop_get:NVNTF \l__template_keytypes_prop
426     \l__template_key_name_tl
427     \l__template_keytype_tl
428     {
429       \_template_split_keytype_arg:V \l__template_keytype_tl
430       \_template_parse_vars_elt_aux:nn {#1} {#3}
431       \prop_remove:NV \l__template_keytypes_prop \l__template_key_name_tl
432     }
433     { \msg_error:nnn { template } { unknown-key } {#2} }
434 }

```

Split off any leading global and they look for the way to implement.

```

435 \cs_new_protected:Npn \_template_parse_vars_elt_aux:nn #1#2
436 {
437   \_template_parse_vars_elt_aux:nw {#1} #2 global global \s__template_stop
438 }
439 \cs_new_protected:Npn \_template_parse_vars_elt_aux:nw
440   #1#2 global #3 global #4 \s__template_stop
441   {
442     \tl_if_blank:nTF {#4}
443     { \_template_parse_vars_elt_aux:nnn {#1} { } {#2} }
444     {
445       \tl_if_blank:nTF {#2}
446       {
447         \_template_parse_vars_elt_aux:nne
448           {#1} { global } { \tl_trim_spaces:n {#3} }
449       }
450       { \msg_error:nnn { template } { bad-variable } { #2 global #3 } }
451     }
452   }
453 \cs_new_protected:Npn \_template_parse_vars_elt_aux:nnn #1#2#3
454 {
455   \str_case:VnF \l__template_keytype_tl
456   {
457     { choice } { \_template_implement_choices:nn {#1} {#3} }
458     { function }
459     {
460       \cs_if_exist:NF #3
461       { \cs_new:Npn #3 { } }
462       \_template_parse_vars_elt_key:nn {#1}
463       {
464         .code:n =
465         {
466           \cs_generate_from_arg_count:NNnn
467             \exp_not:N #3
468             \exp_not:c
469             { cs_ \str_if_eq:nnT {#1} { global } { g } set:Npn }

```

```

470         { \exp_not:V \l__template_keytype_arg_tl }
471         {##1}
472     }
473 }
474 \prop_put:NVn \l__template_vars_prop
475 \l__template_key_name_tl {#2#3}
476 }
477 { instance }
478 {
479     \__template_parse_vars_elt_key:nn {#1}
480     {
481         .code:n =
482         {
483             \exp_not:c
484             { cs_ \str_if_eq:nnT {#1} { global } { g } set:Npn }
485             \exp_not:N #3 { \UseInstance {##1} }
486         }
487     }
488     \prop_put:NVn \l__template_vars_prop
489     \l__template_key_name_tl {#2#3}
490 }
491 }
492 {
493     \tl_if_single:nTF {#3}
494     {
495         \cs_if_exist:NF #3
496         { \use:c { \__template_map_var_type: _new:N } #3 }
497         \__template_parse_vars_elt_key:nn {#1}
498         {
499             . \__template_map_var_type:
500             _ \str_if_eq:nnT {#1} { global } { g } set:N
501             = \exp_not:N #3
502         }
503         \prop_put:NVn \l__template_vars_prop
504         \l__template_key_name_tl {#2#3}
505     }
506     { \msg_error:nnn { template } { bad-variable } {#2#3} }
507 }
508 }
509 \cs_generate_variant:Nn \__template_parse_vars_elt_aux:nnn { nne }
510 \cs_new_protected:Npn \__template_parse_vars_elt_key:nn #1#2
511 {
512     \keys_define:ne { template / #1 }
513     { \l__template_key_name_tl #2 }
514 }

```

(End of definition for __template_parse_vars_elt:nnn and others.)

__template_map_var_type: Turn a “friendly” variable type into an expl3 one.

```

515 \cs_new:Npn \__template_map_var_type:
516 {
517     \str_case:Vn \l__template_keytype_tl
518     {
519         { boolean } { bool }

```



```

520         { commalist } { clist }
521         { integer }   { int }
522         { length }    { dim }
523         { muskip }    { muskip }
524         { real }      { fp }
525         { skip }      { skip }
526         { tokenlist } { t1 }
527     }
528 }

```

(End of definition for `__template_map_var_type:`.)

```

\__template_implement_choices:nn
\__template_implement_choices_default:

```

Implementing choices requires a second key–value loop. So after a little set-up, the standard parser is called.

```

529 \cs_new_protected:Npn \__template_implement_choices:nn #1#2
530 {
531     \clist_set:NV \l__template_tmp_clist \l__template_keytype_arg_tl
532     \prop_put:NVn \l__template_vars_prop \l__template_key_name_tl { }
533     \keys_define:ne { template / #1 } { \l__template_key_name_tl .choice: }
534     \keyval_parse:nnn
535     { \__template_implement_choice_elt:n }
536     { \__template_implement_choice_elt:nnn {#1} }
537     {#2}
538     \prop_get:NVNT \l__template_values_prop \l__template_key_name_tl
539     \l__template_tmp_tl
540     { \__template_implement_choices_default: }
541     \clist_if_empty:NF \l__template_tmp_clist
542     {
543         \clist_map_inline:Nn \l__template_tmp_clist
544         { \msg_error:nnn { template } { choice-not-implemented } {##1} }
545     }
546 }

```

A sanity check for the default value, so that an error is raised now and not when converting to assignments.

```

547 \cs_new_protected:Npn \__template_implement_choices_default:
548 {
549     \tl_set:Ne \l__template_tmp_tl
550     { \l__template_key_name_tl \c_space_tl \l__template_tmp_tl }
551     \prop_if_in:NVF \l__template_vars_prop \l__template_tmp_tl
552     {
553         \tl_set:Ne \l__template_tmp_tl
554         { \l__template_key_name_tl \c_space_tl \l__template_tmp_tl }
555         \prop_if_in:NVF \l__template_vars_prop \l__template_tmp_tl
556         {
557             \prop_get:NVN \l__template_keytypes_prop \l__template_key_name_tl
558             \l__template_tmp_tl
559             \__template_split_keytype_arg:V \l__template_tmp_tl
560             \prop_get:NVN \l__template_values_prop \l__template_key_name_tl
561             \l__template_tmp_tl
562             \msg_error:nnVV { template } { unknown-default-choice }
563             \l__template_key_name_tl
564             \l__template_key_name_tl
565         }
566     }
}

```

```
567 }
```

(End of definition for `__template_implement_choices:nn` and `__template_implement_choices_default:.`)

```
\__template_implement_choice_elt:nnn
\__template_implement_choice_elt_aux:nnn
\__template_implement_choice_elt_aux:n
\__template_implement_choice_elt:n
```

The actual storage of the implementation of a choice is mainly about error checking. The code here ensures that all choices have to have been declared, apart from the special **unknown** choice, which must come last. The code for each choice is stored along with the key name in the variables property list.

```
568 \cs_new_protected:Npn \__template_implement_choice_elt:nnn #1#2#3
569 {
570   \clist_if_empty:NTF \l__template_tmp_clist
571   {
572     \str_if_eq:nnTF {#2} { unknown }
573     { \__template_implement_choice_elt_aux:nnn {#1} {#2} {#3} }
574     { \__template_implement_choice_elt_aux:n {#2} }
575   }
576   {
577     \clist_if_in:NnTF \l__template_tmp_clist {#2}
578     {
579       \clist_remove_all:Nn \l__template_tmp_clist {#2}
580       \__template_implement_choice_elt_aux:nnn {#1} {#2} {#3}
581     }
582     { \__template_implement_choice_elt_aux:n {#2} }
583   }
584 }
585 \cs_new_protected:Npn \__template_implement_choice_elt_aux:n #1
586 {
587   \prop_get:NVN \l__template_keytypes_prop \l__template_key_name_tl
588   \l__template_tmp_tl
589   \__template_split_keytype_arg:V \l__template_tmp_tl
590   \msg_error:nnVn { template } { unknown-choice } \l__template_key_name_tl {#1}
591 }
592 \cs_new_protected:Npn \__template_implement_choice_elt_aux:nnn #1#2#3
593 {
594   \keys_define:ne { template / #1 }
595   { \l__template_key_name_tl / #2 .code:n = { \exp_not:n {#3} } }
596   \tl_set:Ne \l__template_tmp_tl
597   { \l__template_key_name_tl \c_space_tl #2 }
598   \prop_put:NVN \l__template_vars_prop \l__template_tmp_tl {#3}
599 }
600 \cs_new_protected:Npn \__template_implement_choice_elt:n #1
601 {
602   \msg_error:nnVn { template } { choice-requires-code }
603   \l__template_key_name_tl {#1}
604 }
```

(End of definition for `__template_implement_choice_elt:nnn` and others.)

11.7 Editing template defaults

```
\__template_edit_defaults:nnn
```

Editing the template defaults means getting the values back out of the store, then parsing the list of new values before putting the updated list back into storage.

```
605 \cs_new_protected:Npn \__template_edit_defaults:nnn #1#2#3
```

```

606 {
607   \__template_if_keys_exist:nnT {#1} {#2}
608   {
609     \__template_recover_defaults:nn {#1} {#2}
610     \__template_parse_values:nnn {#1} {#2} {#3}
611     \__template_store_defaults:nn {#1} {#2}
612   }
613 }

```

(End of definition for __template_edit_defaults:nnn.)

__template_parse_values:nnn The routine to parse values is the same for both editing a template and setting up an instance. So the code here does only the minimum necessary for reading the values.

```

614 \cs_new_protected:Npn \__template_parse_values:nnn #1#2#3
615 {
616   \__template_recover_keytypes:nn {#1} {#2}
617   \keyval_parse:NNn
618   \__template_parse_values_elt:n \__template_parse_values_elt:nn {#3}
619 }

```

(End of definition for __template_parse_values:nnn.)

__template_parse_values_elt:n Every key needs a value, so this is just an error routine.

```

620 \cs_new_protected:Npn \__template_parse_values_elt:n #1
621 {
622   \bool_set_true:N \l__template_error_bool
623   \msg_error:nnn { template } { key-no-value } {#1}
624 }

```

(End of definition for __template_parse_values_elt:n.)

__template_parse_values_elt:nn To store the value, find the keytype then call the saving function. These need the current key name saved as \l__template_key_name_tl.

```

625 \cs_new_protected:Npn \__template_parse_values_elt:nn #1#2
626 {
627   \tl_set:Nx \l__template_key_name_tl
628   { \tl_trim_spaces:e { \tl_to_str:n {#1} } }
629   \prop_get:NVNTF \l__template_keytypes_prop \l__template_key_name_tl
630   \l__template_tmp_tl
631   { \__template_parse_values_elt_aux:n {#2} }
632   { \msg_error:nnV { template } { unknown-key } \l__template_key_name_tl }
633 }
634 \cs_new_protected:Npn \__template_parse_values_elt_aux:n #1
635 {
636   \__template_split_keytype_arg:V \l__template_tmp_tl
637   \use:c { __template_store_value_ \l__template_keytype_tl :n } {#1}
638 }

```

(End of definition for __template_parse_values_elt:nn and __template_parse_values_elt_aux:n.)

__template_template_set_eq:nnn To copy a template, each of the lists plus the code has to be copied across. To keep this independent of the list storage system, it is all done with two-part shuffles.

```

639 \cs_new_protected:Npn \__template_template_set_eq:nnn #1#2#3
640 {
641   \__template_recover_defaults:nn {#1} {#3}

```

```

642     \__template_store_defaults:nn {#1} {#2}
643     \__template_recover_keytypes:nn {#1} {#3}
644     \__template_store_keytypes:nn {#1} {#2}
645     \__template_recover_vars:nn {#1} {#3}
646     \__template_store_vars:nn {#1} {#2}
647     \cs_if_exist:cT { \c__template_code_root_tl #1 / #2 }
648       { \msg_info:nnnn { template } { declare-template-code } {#1} {#2} }
649     \cs_gset_eq:cc { \c__template_code_root_tl #1 / #2 }
650       { \c__template_code_root_tl #1 / #3 }
651   }

```

(End of definition for __template_template_set_eq:nnn.)

11.8 Creating instances of templates

```

\__template_declare_instance:nnnn
\__template_declare_instance_aux:nnnn

```

Making an instance has two distinct parts. First, the keys given are parsed to transfer the values into the structured data format used internally. This allows the default and given values to be combined with no repetition. In the second step, the structured data is converted to pre-defined variable assignments, and these are stored in the function for the instance.

```

652 \cs_new_protected:Npn \__template_declare_instance:nnnn #1#2#3#4
653 {
654   \__template_execute_if_code_exist:nnT {#1} {#2}
655   {
656     \__template_recover_defaults:nn {#1} {#2}
657     \__template_recover_vars:nn {#1} {#2}
658     \__template_declare_instance_aux:nnnn {#1} {#2} {#3} {#4}
659   }
660 }
661 \cs_new_protected:Npn \__template_declare_instance_aux:nnnn #1#2#3#4
662 {
663   \bool_set_false:N \l__template_error_bool
664   \__template_parse_values:nnn {#1} {#2} {#4}
665   \bool_if:NF \l__template_error_bool
666   {
667     \prop_put:Nnn \l__template_values_prop { from-template } {#2}
668     \__template_store_values:nn {#1} {#3}
669     \__template_convert_to_assignments:
670     \cs_if_exist:cT { \c__template_instances_root_tl #1 / #3 }
671       { \msg_info:nnnn { template } { declare-instance } {#3} {#1} }
672     \cs_set_protected:cpe { \c__template_instances_root_tl #1 / #3 }
673     {
674       \exp_not:N \__template_assignments_push:n
675       { \exp_not:V \l__template_assignments_tl }
676       \exp_not:c { \c__template_code_root_tl #1 / #2 }
677     }
678   }
679 }

```

(End of definition for __template_declare_instance:nnnn and __template_declare_instance_aux:nnnn.)

```

\__template_instance_set_eq:nnn

```

Copy-paste an instance.

```

680 \cs_new_protected:Npn \__template_instance_set_eq:nnn #1#2#3

```

```

681 {
682   \__template_if_instance_exist:nnTF {#1} {#3}
683   {
684     \__template_recover_values:nn {#1} {#3}
685     \__template_store_values:nn {#1} {#2}
686     \cs_if_exist:cT { \c__template_instances_root_tl #1 / #2 }
687     { \msg_info:nnnn { template } { declare-instance } {#2} {#1} }
688     \cs_set_eq:cc { \c__template_instances_root_tl #1 / #2 }
689     { \c__template_instances_root_tl #1 / #3 }
690   }
691   { \msg_error:nnnn { template } { unknown-instance } {#1} {#3} }
692 }

```

(End of definition for __template_instance_set_eq:nnn.)

__template_edit_instance:nnn
 __template_edit_instance_aux:nnnnn
 __template_edit_instance_aux:nVnnn

Editing an instance is almost identical to declaring one. The only variation is the source of the values to use. When editing, they are recovered from the previous instance run.

```

693 \cs_new_protected:Npn \__template_edit_instance:nnn #1#2#3
694 {
695   \__template_if_instance_exist:nnTF {#1} {#2}
696   {
697     \__template_recover_values:nn {#1} {#2}
698     \prop_get:NnN \l__template_values_prop { from-template }
699     \l__template_tmp_tl
700     \__template_edit_instance_aux:nVnn
701     {#1} \l__template_tmp_tl {#2} {#3}
702   }
703   { \msg_error:nnnn { template } { unknown-instance } {#1} {#2} }
704 }
705 \cs_new_protected:Npn \__template_edit_instance_aux:nnnn #1#2#3#4
706 {
707   \__template_recover_vars:nn {#1} {#2}
708   \__template_declare_instance_aux:nnnn {#1} {#2} {#3} {#4}
709 }
710 \cs_generate_variant:Nn \__template_edit_instance_aux:nnnn { nV }

```

(End of definition for __template_edit_instance:nnn and __template_edit_instance_aux:nnnnn.)

__template_convert_to_assignments:
 __template_convert_to_assignments_aux:n
 __template_convert_to_assignments_aux:nn
 __template_convert_to_assignments_aux:nV

The idea on converting to a set of assignments is to loop over each key, so that the loop order follows the declaration order of the keys. This is done using a sequence as property lists are not “ordered”.

```

711 \cs_new_protected:Npn \__template_convert_to_assignments:
712 {
713   \tl_clear:N \l__template_assignments_tl
714   \seq_map_function:NN \l__template_key_order_seq
715   \__template_convert_to_assignments_aux:n
716 }
717 \cs_new_protected:Npn \__template_convert_to_assignments_aux:n #1
718 {
719   \prop_get:NnN \l__template_keytypes_prop {#1} \l__template_tmp_tl
720   \__template_convert_to_assignments_aux:nV {#1} \l__template_tmp_tl
721 }

```

The second auxiliary function actually does the work. The arguments here are the key name (#1) and the keytype (#2). From those, the value to assign and the name of the appropriate variable are recovered. A bit of work is then needed to sort out keytypes with arguments (for example instances), and to look for global assignments. Once that is done, a hand-off can be made to the handler for the relevant keytype.

```

722 \cs_new_protected:Npn \__template_convert_to_assignments_aux:nn #1#2
723 {
724   \prop_get:NnNT \l__template_values_prop {#1} \l__template_value_tl
725   {
726     \prop_get:NnNTF \l__template_vars_prop {#1} \l__template_var_tl
727     {
728       \__template_split_keytype_arg:n {#2}
729       \str_if_eq:VnF \l__template_keytype_tl { choice }
730       {
731         \str_if_eq:VnF \l__template_keytype_tl { code }
732         { \__template_find_global: }
733       }
734       \tl_set:Nn \l__template_key_name_tl {#1}
735       \cs_if_exist_use:cF { __template_assign_ \l__template_keytype_tl : }
736       { \__template_assign_variable: }
737     }
738     { \msg_error:nnn { template } { unknown-attribute } {#1} }
739   }
740 }
741 \cs_generate_variant:Nn \__template_convert_to_assignments_aux:nn { nV }

```

(End of definition for `__template_convert_to_assignments:`, `__template_convert_to_assignments_aux:n`, and `__template_convert_to_assignments_aux:nn`.)

`__template_find_global:` Global assignments should have the phrase `global` at the front. This is pretty easy to find: no other error checking, though.

```

742 \cs_new_protected:Npn \__template_find_global:
743 {
744   \bool_set_false:N \l__template_global_bool
745   \tl_if_in:onT \l__template_var_tl { global }
746   {
747     \exp_after:wN \__template_find_global_aux:w \l__template_var_tl \s__template_stop
748   }
749 }
750 \cs_new_protected:Npn \__template_find_global_aux:w #1 global #2 \s__template_stop
751 {
752   \tl_set:Nn \l__template_var_tl {#2}
753   \bool_set_true:N \l__template_global_bool
754 }

```

(End of definition for `__template_find_global:` and `__template_find_global_aux:w`.)

11.9 Using templates directly

`__template_use_template:nnn` Directly use a template with a particular parameter setting. This is also picked up if used in a nested fashion inside a parameter list. The idea is essentially the same as creating an instance, just with no saving of the result.

```

755 \cs_new_protected:Npn \__template_use_template:nnn #1#2#3

```

```

756 {
757   \__template_execute_if_code_exist:nnT {#1} {#2}
758   {
759     \__template_recover_defaults:nn {#1} {#2}
760     \__template_recover_vars:nn {#1} {#2}
761     \__template_parse_values:nnn {#1} {#2} {#3}
762     \__template_convert_to_assignments:
763     \use:c { \c__template_code_root_tl #1 / #2 }
764   }
765 }

```

(End of definition for __template_use_template:nnn.)

11.10 Assigning values to variables

__template_assign_boolean: Setting a Boolean value is slightly different to everything else as the value can be used to work out which `set` function to call. As long as there is no need to recover things from another variable, everything is pretty easy. If there is, then we need to allow for the fact that the recovered value here will *not* be expandable, so needs to be converted to something that is.

```

766 \cs_new_protected:Npn \__template_assign_boolean:
767 {
768   \bool_if:NTF \l__template_global_bool
769   { \__template_assign_boolean_aux:n { bool_gset } }
770   { \__template_assign_boolean_aux:n { bool_set } }
771 }
772 \cs_new_protected:Npn \__template_assign_boolean_aux:n #1
773 {
774   \__template_if_key_value:VTF \l__template_value_tl
775   {
776     \__template_key_to_value:
777     \tl_put_right:Ne \l__template_assignments_tl
778     {
779       \exp_not:c { #1 _eq:NN }
780       \exp_not:V \l__template_var_tl
781       \exp_not:V \l__template_value_tl
782     }
783   }
784   {
785     \tl_put_right:Ne \l__template_assignments_tl
786     {
787       \exp_not:c { #1 _ \l__template_value_tl :N }
788       \exp_not:V \l__template_var_tl
789     }
790   }
791 }

```

(End of definition for __template_assign_boolean: and __template_assign_boolean_aux:n.)

__template_assign_choice: The idea here is to find either the choice as-given or else the special `unknown` choice, and to copy the appropriate code across.

```

\__template_assign_choice_aux:nF
\__template_assign_choice_aux:eF
792 \cs_new_protected:Npn \__template_assign_choice:
793 {
794   \__template_assign_choice_aux:eF

```

```

795     { \l__template_key_name_tl \c_space_tl \l__template_value_tl }
796   {
797     \__template_assign_choice_aux:eF
798     { \l__template_key_name_tl \c_space_tl unknown }
799     {
800       \prop_get:NVN \l__template_keytypes_prop \l__template_key_name_tl
801       \l__template_tmp_tl
802       \__template_split_keytype_arg:V \l__template_tmp_tl
803       \msg_error:nnVV { template } { unknown-choice }
804       \l__template_key_name_tl
805       \l__template_value_tl
806     }
807   }
808 }
809 \cs_new_protected:Npn \__template_assign_choice_aux:nF #1
810 {
811   \prop_get:NnNTF \l__template_vars_prop {#1} \l__template_tmp_tl
812   { \tl_put_right:NV \l__template_assignments_tl \l__template_tmp_tl }
813 }
814 \cs_generate_variant:Nn \__template_assign_choice_aux:nF { e }

```

(End of definition for __template_assign_choice: and __template_assign_choice_aux:nF.)

__template_assign_function: This looks a bit messy but is only actually one function.

```

\__template_assign_function_aux:N
815 \cs_new_protected:Npn \__template_assign_function:
816 {
817   \bool_if:NTF \l__template_global_bool
818   { \__template_assign_function_aux:N \cs_gset:Npn }
819   { \__template_assign_function_aux:N \cs_set:Npn }
820 }
821 \cs_new_protected:Npn \__template_assign_function_aux:N #1
822 {
823   \tl_put_right:Ne \l__template_assignments_tl
824   {
825     \cs_generate_from_arg_count:NNnn
826     \exp_not:V \l__template_var_tl
827     \exp_not:N #1
828     { \exp_not:V \l__template_keytype_arg_tl }
829     { \exp_not:V \l__template_value_tl }
830   }
831 }

```

(End of definition for __template_assign_function: and __template_assign_function_aux:N.)

__template_assign_instance: Using an instance means adding the appropriate function creation to the tl. No checks are made at this stage, so if the instance is not valid then errors will arise later.

```

\__template_assign_instance_aux:N
832 \cs_new_protected:Npn \__template_assign_instance:
833 {
834   \bool_if:NTF \l__template_global_bool
835   { \__template_assign_instance_aux:N \cs_gset_protected:Npn }
836   { \__template_assign_instance_aux:N \cs_set_protected:Npn }
837 }
838 \cs_new_protected:Npn \__template_assign_instance_aux:N #1
839 {

```



```

840     \tl_put_right:Ne \l__template_assignments_tl
841     {
842         \exp_not:N #1 \exp_not:V \l__template_var_tl
843         {
844             \__template_use_instance:nn
845             { \exp_not:V \l__template_keytype_arg_tl }
846             { \exp_not:V \l__template_value_tl }
847         }
848     }
849 }

```

(End of definition for __template_assign_instance: and __template_assign_instance_aux:N.)

__template_assign_variable: A general-purpose function for all of the other assignments. As long as the value is not coming from another variable, the stored value is simply transferred for output. We use V-type expansion for the \KeyValue case: for token lists this is essential, whilst for register-based variables, it does no harm and avoids needing a low-level test.

```

850 \cs_new_protected:Npn \__template_assign_variable:
851 {
852     \exp_args:Ne \__template_assign_variable:n
853     {
854         \__template_map_var_type:
855         -
856         \bool_if:NT \l__template_global_bool { g }
857         set:N
858     }
859 }

```

Notice we need a V-type variant for each (g)set operation here: these need to be provided by expl3.

```

860 \cs_new_protected:Npn \__template_assign_variable:n #1
861 {
862     \__template_if_key_value:VTF \l__template_value_tl
863     {
864         \__template_key_to_value:
865         \tl_put_right:Ne \l__template_assignments_tl
866         {
867             \exp_not:c { #1 V } \exp_not:V \l__template_var_tl
868             \exp_not:V \l__template_value_tl
869         }
870     }
871     {
872         \tl_put_right:Ne \l__template_assignments_tl
873         {
874             \exp_not:c { #1 n } \exp_not:V \l__template_var_tl
875             { \exp_not:V \l__template_value_tl }
876         }
877     }
878 }

```

(End of definition for __template_assign_variable: and __template_assign_variable:n.)

__template_key_to_value: The idea here is to recover the attribute value of another key. To do that, the marker is removed and a look up takes place. If this is successful, then the name of the variable of the attribute is returned. This assumes that the value will be used in context where it

will be converted to a value, for example when setting a number. There is also a need to check in case the copied value happens to be global.

```

879 \cs_new_protected:Npn \__template_key_to_value:
880 { \exp_after:wN \__template_key_to_value_auxi:w \l__template_value_tl }
881 \cs_new_protected:Npn \__template_key_to_value_auxi:w \KeyValue #1
882 {
883   \tl_set:Ne \l__template_tmp_tl { \tl_trim_spaces:e { \tl_to_str:n {#1} } }
884   \prop_get:NVNTF \l__template_vars_prop \l__template_tmp_tl
885     \l__template_value_tl
886   {
887     \exp_after:wN \__template_key_to_value_auxii:w \l__template_value_tl
888     \s__template_mark global \q__template_nil \s__template_stop
889   }
890   { \msg_error:nnV { template } { unknown-attribute } \l__template_tmp_tl }
891 }
892 \cs_new_protected:Npn \__template_key_to_value_auxii:w #1 global #2#3 \s__template_stop
893 {
894   \__template_quark_if_nil:NF #2
895   { \tl_set:Nn \l__template_value_tl {#2} }
896 }

```

(End of definition for `__template_key_to_value:`, `__template_key_to_value_auxi:w`, and `__template_key_to_value_auxii:w`.)

11.11 Using instances

```

\__template_use_instance:nn
  \__template_use_instance_aux:nNnnn
    \__template_use_instance_aux:nn

```

Using an instance is just a question of finding the appropriate function. If nothing is found, an error is raised. One complication is that if the first token of argument #2 is `\UseTemplate` then that is also valid. There is an error-test to make sure that the types agree, and if so the template is used directly.

```

897 \cs_new_protected:Npn \__template_use_instance:nn #1#2
898 {
899   \__template_if_use_template:nTF {#2}
900   { \__template_use_instance_aux:nNnnn {#1} #2 }
901   { \__template_use_instance_aux:nn {#1} {#2} }
902 }
903 \cs_new_protected:Npn \__template_use_instance_aux:nNnnn #1#2#3#4#5
904 {
905   \str_if_eq:nnTF {#1} {#3}
906   { \__template_use_template:nnn {#3} {#4} {#5} }
907   { \msg_error:nnnn { template } { type-mismatch } {#1} {#3} }
908 }
909 \cs_new_protected:Npn \__template_use_instance_aux:nn #1#2
910 {
911   \__template_if_instance_exist:nnTF {#1} {#2}
912   { \use:c { \c__template_instances_root_tl #1 / #2 } }
913   { \msg_error:nnnn { template } { unknown-instance } {#1} {#2} }
914 }

```

(End of definition for `__template_use_instance:nn`, `__template_use_instance_aux:nNnnn`, and `__template_use_instance_aux:nn`.)

11.12 Assignment manipulation

A few functions to transfer assignments about, as this is needed by \AssignTemplateKeys.

_template_assignments_pop: To actually use the assignments.

```

915 \cs_new:Npn \_template_assignments_pop: { \l__template_assignments_tl }

```

(End of definition for _template_assignments_pop:.)

_template_assignments_push:n Here, the assignments are stored for later use.

```

916 \cs_new_protected:Npn \_template_assignments_push:n #1
917 { \tl_set:Nn \l__template_assignments_tl {#1} }

```

(End of definition for _template_assignments_push:n.)

11.13 Showing templates and instances

_template_show_code:nn Showing the code for a template is just a translation of \cs_show:c.

```

918 \cs_new_protected:Npn \_template_show_code:nn #1#2
919 { \cs_show:c { \c__template_code_root_tl #1 / #2 } }

```

(End of definition for _template_show_code:nn.)

_template_show_defaults:nn A modified version of the property-list printing code, such that the output refers to templates and instances rather than to the underlying structures.

```

920 \cs_new_protected:Npn \_template_show_defaults:nn #1#2
921 {
922   \_template_if_keys_exist:nnT {#1} {#2}
923   {
924     \_template_recover_defaults:nn {#1} {#2}
925     \_template_show:Nnnn \l__template_values_prop
926       {#1} {#2} { default~values }
927   }
928 }
929 \cs_new_protected:Npn \_template_show_keytypes:nn #1#2
930 {
931   \_template_if_keys_exist:nnT {#1} {#2}
932   {
933     \_template_recover_keytypes:nn {#1} {#2}
934     \_template_show:Nnnn \l__template_keytypes_prop
935       {#1} {#2} { interface }
936   }
937 }
938 \cs_new_protected:Npn \_template_show_vars:nn #1#2
939 {
940   \_template_execute_if_code_exist:nnT {#1} {#2}
941   {
942     \_template_recover_vars:nn {#1} {#2}
943     \_template_show:Nnnn \l__template_vars_prop
944       {#1} {#2} { variable~mapping }
945   }
946 }
947 \cs_new_protected:Npn \_template_show:Nnnn #1#2#3#4
948 {

```

```

949 \msg_show:nneeee { template } { show-attribute }
950 { \tl_to_str:n {#2} }
951 { \tl_to_str:n {#3} }
952 { \tl_to_str:n {#4} }
953 { \prop_map_function:NN #1 \msg_show_item_unbraced:nn }
954 }

```

(End of definition for `__template_show_defaults:nn` and others.)

`__template_show_values:nn` Instance values are a little more complex, as is the template to consider.

```

955 \cs_new_protected:Npn \__template_show_values:nn #1#2
956 {
957   \__template_if_instance_exist:nnT {#1} {#2}
958   {
959     \__template_recover_values:nn {#1} {#2}
960     \msg_show:nneee { template } { show-values }
961     { \tl_to_str:n {#1} }
962     { \tl_to_str:n {#2} }
963     {
964       \prop_map_function:NN \l__template_values_prop
965       \msg_show_item_unbraced:nn
966     }
967   }
968 }

```

(End of definition for `__template_show_values:nn`.)

11.14 Messages

The text for error messages: short and long text for all of them.

```

969 \msg_new:nnnn { template } { argument-number-mismatch }
970 { Template~type~'~#1'~takes~#2~argument(s). }
971 {
972   Templates-of~type~'~#1'~require~#2~argument(s).\
973   You-have~tried-to-make-a-template-for~'~#1'~
974   with~#3~argument(s),~which-is-not-possible:-
975   the~number-of~arguments~must~agree.
976 }
977 \msg_new:nnnn { template } { bad-number-of-arguments }
978 { Bad-number-of~arguments~for~template~type~'~#1'. }
979 {
980   A-template-may-accept~between~0~and~9~arguments.\
981   You-asked-to-use~#2~arguments:-this-is-not-supported.
982 }
983 \msg_new:nnnn { template } { bad-variable }
984 { Incorrect-variable-description~'~#1'. }
985 {
986   The~argument~'~#1'~is-not-of-the~form \
987   ~~~<variable>'\
988   ~or~\
989   ~~~global-<variable>'. \
990   It-must-be-given-in-one-of~these-formats-to-be-used-in-a-template.
991 }
992 \msg_new:nnnn { template } { choice-not-implemented }

```

```

993 { The-choice-~'#1'~has-no-implementation. }
994 {
995     Each-choice-listed-in-the-interface-for-a-template-must-
996     have-an-implementation.
997 }
998 \msg_new:nnnn { template } { choice-no-code }
999 { The-choice-~'#1'~requires-implementation-details. }
1000 {
1001     When-creating-template-code-using-\DeclareTemplateCode,~
1002     each-choice-name-must-have-an-associated-implementation.\\
1003     This-should-be-given-after-a-~'= '~sign:~LaTeX-did-not-find-one.
1004 }
1005 \msg_new:nnnn { template } { choice-requires-code }
1006 { The-choice-~'#2'~for-key-~'#1'~requires-an-implementation. }
1007 {
1008     You-should-have-put:\\
1009     \ \ #1::~choice-~{~#2 = <code> ~} \\
1010     but-LaTeX-did-not-find-any-<code>.
1011 }
1012 \msg_new:nnnn { template } { duplicate-key-interface }
1013 { Key-~'#1'~appears-twice-in-interface-definition~\msg_line_context:. }
1014 {
1015     Each-key-can-only-have-one-interface-declared-in-a-template.\\
1016     LaTeX-found-two-interfaces-for-~'#1'.
1017 }
1018 \msg_new:nnnn { template } { keytype-requires-argument }
1019 { The-key-type-~'#1'~requires-an-argument~\msg_line_context:. }
1020 {
1021     You-should-have-put:\\
1022     \ \ <key-name>::~~#1-~{~<argument>~} \\
1023     but-LaTeX-did-not-find-an-<argument>.
1024 }
1025 \msg_new:nnnn { template } { invalid-keytype }
1026 { The-key-~'#1'~is-missing-a-key-type~\msg_line_context:. }
1027 {
1028     Each-key-in-a-template-requires-a-key-type,~given-in-the-form:\\
1029     \ \ <key>::~~<key-type>\\
1030     LaTeX-could-not-find-a~<key-type>-in-your-input.
1031 }
1032 \msg_new:nnnn { template } { key-no-value }
1033 { The-key-~'#1'~has-no-value~\msg_line_context:. }
1034 {
1035     When-creating-an-instance-of-a-template-
1036     every-key-listed-must-include-a-value:\\
1037     \ \ <key>::~~<value>
1038 }
1039 \msg_new:nnnn { template } { key-no-variable }
1040 { The-key-~'#1'~requires-implementation-details~\msg_line_context:. }
1041 {
1042     When-creating-template-code-using-\DeclareTemplateCode,~
1043     each-key-name-must-have-an-associated-implementation.\\
1044     This-should-be-given-after-a-~'= '~sign:~LaTeX-did-not-find-one.
1045 }
1046 \msg_new:nnnn { template } { key-not-implemented }

```

```

1047 { Key~'#1'~has~no~implementation~\msg_line_context:. }
1048 {
1049   The~definition~of~key~implementations~for~template~'#2'~
1050   of~template~type~'#3'~does~not~include~any~details~for~key~'#1'.\\
1051   The~key~was~declared~in~the~interface~definition,~
1052   and~so~an~implementation~is~required.
1053 }
1054 \msg_new:nnnn { template } { missing-keytype }
1055 { The~key~'#1'~is~missing~a~key~type~\msg_line_context:. }
1056 {
1057   Key~interface~definitions~should~be~of~the~form\\
1058   \ \ #1~::~<key-type>\\
1059   but~LaTeX~could~not~find~a~<key-type>.
1060 }
1061 \msg_new:nnnn { template } { no-template-code }
1062 {
1063   The~template~'#2'~of~type~'#1'~is~unknown~
1064   or~has~no~implementation.
1065 }
1066 {
1067   There~is~no~code~available~for~the~template~name~given.\\
1068   This~should~be~given~using~\DeclareTemplateCode.
1069 }
1070 \msg_new:nnnn { template } { type-already-defined }
1071 { Template~type~'#1'~already~defined. }
1072 {
1073   You~have~used~\NewTemplateType~
1074   with~a~template~type~that~has~already~been~defined.
1075 }
1076 \msg_new:nnnn { template } { type-mismatch }
1077 { Template~types~'#1'~and~'#2'~do~not~agree. }
1078 {
1079   You~are~trying~to~use~a~template~directly~with~\UseInstance
1080   (or~a~similar~function),~but~the~template~types~do~not~match.
1081 }
1082 \msg_new:nnnn { template } { unknown-attribute }
1083 { The~template~attribute~'#1'~is~unknown. }
1084 {
1085   There~is~a~definition~in~the~current~template~reading\\
1086   \ \ \token_to_str:N \KeyValue {~#1~} \\
1087   but~there~is~no~key~called~'#1'.
1088 }
1089 \msg_new:nnnn { template } { unknown-choice }
1090 { The~choice~'#2'~was~not~declared~for~key~'#1'. }
1091 {
1092   The~key~'#1'~takes~a~fixed~list~of~choices~
1093   and~this~list~does~not~include~'#2'.
1094 }
1095 \msg_new:nnnn { template } { unknown-default-choice }
1096 { The~default~choice~'#2'~was~not~declared~for~key~'#1'. }
1097 {
1098   The~key~'#1'~takes~a~fixed~list~of~choices~
1099   and~this~list~does~not~include~'#2'.
1100 }

```

```

1101 \msg_new:nnnn { template } { unknown-instance }
1102 { The~instance~'#2'~of~type~'#1'~is~unknown. }
1103 {
1104     You~have~asked~to~use~an~instance~'#2',~
1105     but~this~has~not~been~created.
1106 }
1107 \msg_new:nnnn { template } { unknown-key }
1108 { Unknown~template~key~'#1'. }
1109 {
1110     The~key~'#1'~was~not~declared~in~the~interface~
1111     for~the~current~template.
1112 }
1113 \msg_new:nnnn { template } { unknown-keytype }
1114 { The~key~type~'#1'~is~unknown. }
1115 {
1116     Valid~key~types~are:\\
1117     --boolean;\\
1118     --choice;\\
1119     --commalist;\\
1120     --function;\\
1121     --instance;\\
1122     --integer;\\
1123     --length;\\
1124     --muskip;\\
1125     --real;\\
1126     --skip;\\
1127     --tokenlist.
1128 }
1129 \msg_new:nnnn { template } { unknown-type }
1130 { The~template~type~'#1'~is~unknown. }
1131 {
1132     A~template~type~needs~to~be~defined~with~\NewTemplateType
1133     prior~to~using~it.
1134 }
1135 \msg_new:nnnn { template } { unknown-template }
1136 { The~template~'#2'~of~type~'#1'~is~unknown. }
1137 {
1138     No~interface~has~been~declared~for~a~template~
1139     '#2'~of~template~type~'#1'.
1140 }

```

Information messages only have text: more text should not be needed.

```

1141 \msg_new:nnn { template } { declare-instance }
1142 { Declaring~instance~'#1'~of~type~'#2'~\msg_line_context:. }
1143 \msg_new:nnn { template } { declare-template-code }
1144 { Declaring~code~for~template~'#2'~of~template~type~'#1'~\msg_line_context:. }
1145 \msg_new:nnn { template } { declare-template-interface }
1146 {
1147     Declaring~interface~for~template~'#2'~of~template~type~'#1'~
1148     \msg_line_context:.
1149 }
1150 \msg_new:nnn { template } { declare-type }
1151 { Declaring~template~type~'#1'~taking~#2~argument(s)~\msg_line_context:. }
1152 \msg_new:nnn { template } { show-attribute }
1153 {

```

```

1154     The~template~'~#2'~of~type~'~#1'~has~
1155     \tl_if_empty:nTF {#4} { no~#3. } { #3 : #4 }
1156   }
1157   \msg_new:nnn { template } { show-values }
1158   {
1159     The~instance~'~#2'~of~type~'~#1'~has~
1160     \tl_if_empty:nTF {#3} { no~values. } { values: #3 }
1161   }

```

Also add template to the LaTeX messages.

```

1162 \prop_gput:Nnn \g_msg_module_type_prop { template } { LaTeX }

```

11.15 User functions

All simple translations.

```

\NewTemplateType
\DeclareTemplateInterface 1163 \cs_new_protected:Npn \NewTemplateType #1#2
\DeclareTemplateCode      1164 { \__template_define_type:nn {#1} {#2} }
\DeclareTemplateCopy      1165 \cs_new_protected:Npn \DeclareTemplateInterface #1#2#3#4
\EditTemplateDefaults     1166 { \__template_declare_template_keys:nnnn {#1} {#2} {#3} {#4} }
\UseTemplate              1167 \cs_new_protected:Npn \DeclareTemplateCode #1#2#3#4#5
\DeclareInstance          1168 { \__template_declare_template_code:nnnnn {#1} {#2} {#3} {#4} {#5} }
\DeclareInstanceCopy      1169 \cs_new_protected:Npn \DeclareTemplateCopy #1#2#3
\EditInstance             1170 { \__template_template_set_eq:nnn {#1} {#2} {#3} }
\UseInstance              1171 \cs_new_protected:Npn \EditTemplateDefaults #1#2#3
                          1172 { \__template_edit_defaults:nnn {#1} {#2} {#3} }
                          1173 \cs_new_protected:Npn \UseTemplate #1#2#3
                          1174 { \__template_use_template:nnn {#1} {#2} {#3} }
                          1175 \cs_new_protected:Npn \DeclareInstance #1#2#3#4
                          1176 { \__template_declare_instance:nnnn {#1} {#3} {#2} {#4} }
                          1177 \cs_new_protected:Npn \DeclareInstanceCopy #1#2#3
                          1178 { \__template_instance_set_eq:nnn {#1} {#2} {#3} }
                          1179 \cs_new_protected:Npn \EditInstance #1#2#3
                          1180 { \__template_edit_instance:nnn {#1} {#2} {#3} }
                          1181 \cs_new_protected:Npn \UseInstance #1#2
                          1182 { \__template_use_instance:nn {#1} {#2} }

```

(End of definition for \NewTemplateType and others. These functions are documented on page 3.)

The show functions are again just translation.

```

\ShowTemplateCode          1183 \cs_new_protected:Npn \ShowTemplateCode #1#2
\ShowTemplateDefaults      1184 { \__template_show_code:nn {#1} {#2} }
\ShowTemplateInterface     1185 \cs_new_protected:Npn \ShowTemplateDefaults #1#2
\ShowTemplateVariables     1186 { \__template_show_defaults:nn {#1} {#2} }
\ShowInstanceValues       1187 \cs_new_protected:Npn \ShowTemplateInterface #1#2
                          1188 { \__template_show_keytypes:nn {#1} {#2} }
                          1189 \cs_new_protected:Npn \ShowTemplateVariables #1#2
                          1190 { \__template_show_vars:nn {#1} {#2} }
                          1191 \cs_new_protected:Npn \ShowInstanceValues #1#2
                          1192 { \__template_show_values:nn {#1} {#2} }

```

(End of definition for \ShowTemplateCode and others. These functions are documented on page 10.)

More direct translation.

```

\IfInstanceExistsT        1193 \cs_new:Npn \IfInstanceExistsTF #1#2
\IfInstanceExistsF
\IfInstanceExistsTF

```



```

1194 { \__template_if_instance_exist:nnTF {#1} {#2} }
1195 \cs_new:Npn \IfInstanceExistsT #1#2
1196 { \__template_if_instance_exist:nnT {#1} {#2} }
1197 \cs_new:Npn \IfInstanceExistsF #1#2
1198 { \__template_if_instance_exist:nnF {#1} {#2} }

```

(End of definition for \IfInstanceExistsT, \IfInstanceExistsF, and \IfInstanceExistsTF. These functions are documented on page 8.)

\KeyValue Simply dump the argument when executed: this should not happen.

```

1199 \cs_new_protected:Npn \KeyValue #1 {#1}

```

(End of definition for \KeyValue. This function is documented on page 4.)

\AssignTemplateKeys A short call to use a token register by proxy.

```

1200 \cs_new_protected:Npn \AssignTemplateKeys { \__template_assignments_pop: }

```

(End of definition for \AssignTemplateKeys. This function is documented on page 5.)

\SetKnownTemplateKeys A friendly wrapper, with some speed up for the common case of the third argument being empty.
\SetTemplateKeys

```

1201 \cs_new_protected:Npn \SetKnownTemplateKeys #1#2#3
1202 {
1203   \tl_if_empty:oTF {#3}
1204   {
1205     \tl_set_eq:NN \UnusedTemplateKeys \c_empty_tl
1206   }
1207   {
1208     \keys_set_known:noN { template / #1 / #2 } {#3} \UnusedTemplateKeys
1209   }
1210 }
1211 \cs_new_protected:Npn \SetTemplateKeys #1#2#3
1212 {
1213   \tl_if_empty:oF {#3}
1214   {
1215     \keys_set:no { template / #1 / #2 } {#3}
1216   }
1217 }
1218 \tl_new:N \UnusedTemplateKeys

```

(End of definition for \SetKnownTemplateKeys and \SetTemplateKeys. These functions are documented on page 6.)

```

1219 <latexrelease>\IncludeInRelease{0000/00/00}{lttemplates}%
1220 <latexrelease>          {Prototype~document~commands}%
1221 <latexrelease>
1222 <latexrelease>\EndModuleRelease
1223 \ExplSyntaxOff
1224 </2ekernel | latexrelease>

```

We need to stop DocStrip treating @@ in a special way at this point.

```

1225 <@@= >

```

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols	
\\	972, 980, 986, 987, 988, 989, 1002, 1008, 1009, 1015, 1021, 1022, 1028, 1029, 1036, 1043, 1050, 1057, 1058, 1067, 1085, 1086, 1116, 1117, 1118, 1119, 1120, 1121, 1122, 1123, 1124, 1125, 1126
_	1009, 1022, 1029, 1037, 1058, 1086
A	
\AssignTemplateKeys	1200, 389, 394, 5
B	
bool commands:	
\bool_if:NTF	856, 222, 228, 665, 768, 817, 834
\bool_new:N	21, 22
\bool_set_false:N	277, 663, 744
\bool_set_true:N	248, 290, 622, 753
\l_tmpa_bool	5
C	
\c	385
\caption	2
clist commands:	
\clist_if_empty:NTF	541, 570
\clist_if_in:NnTF	266, 577
\clist_map_inline:Nn	543
\clist_new:N	32
\clist_remove_all:Nn	579
\clist_set:Nn	531
\l_tmpa_clist	5
cs commands:	
\cs_generate_from_arg_count:NNnn	404, 466, 825
\cs_generate_variant:Nn	64, 349, 509, 710, 741, 814
\cs_gset:Npn	818
\cs_gset_eq:NN	649
\cs_gset_protected:Npn	406, 835
\cs_if_exist:NTF	54, 60, 73, 86, 402, 460, 495, 647, 670, 686
\cs_if_exist_use:NTF	735
\cs_new:Npn	915, 1193, 1195, 1197, 350, 351, 461, 515
\cs_new_eq:NN	356, 357, 358, 373
\cs_new_protected:Npe	275
\cs_new_protected:Npn	850, 860, 879, 881, 892, 897, 903, 909,
	916, 918, 920, 929, 938, 947, 955, 1163, 1165, 1167, 1169, 1171, 1173, 1175, 1177, 1179, 1181, 1183, 1185, 1187, 1189, 1191, 1199, 1200, 1201, 1211, 42, 52, 58, 65, 71, 96, 104, 120, 128, 136, 146, 162, 172, 182, 188, 203, 219, 241, 253, 270, 296, 320, 352, 354, 359, 361, 363, 365, 367, 369, 371, 374, 400, 408, 419, 421, 435, 439, 453, 510, 529, 547, 568, 585, 592, 600, 605, 614, 620, 625, 634, 639, 652, 661, 680, 693, 705, 711, 717, 722, 742, 750, 755, 766, 772, 792, 809, 815, 821, 832, 838
\cs_set:Npn	328, 819
\cs_set_eq:NN	688
\cs_set_protected:Npe	672
\cs_set_protected:Npn	324, 836
\cs_show:N	919, 35
D	
debug commands:	
\debug_resume:	102, 118, 126, 134
\debug_suspend:	98, 106, 122, 130
\DeclareInstance	1163, 8
\DeclareInstanceCopy	1163, 8
\DeclareTemplateCode	1001, 1042, 1068, 1163, 5
\DeclareTemplateCopy	1163, 7
\DeclareTemplateInterface	1163, 3
dim commands:	
\dim_eval:n	364
\dim_new:N	33
\l_tmpa_dim	5
E	
\EditInstance	1163, 9
\EditTemplateDefaults	1163, 9
\EndModuleRelease	1222
exp commands:	
\exp_after:wN	880, 887, 285, 747
\exp_args:Ne	852
\exp_not:N	867, 874, 277, 278, 279, 280, 290, 296, 299, 301, 302, 306, 308, 311, 316, 467, 468, 483, 485, 501, 674, 676, 779, 787, 827, 842
\exp_not:n	867, 868, 874, 875, 282, 470, 595, 675, 780, 781, 788, 826, 828, 829, 842, 845, 846

\ExplSyntaxOff	1223	1012, 1018, 1025, 1032, 1039, 1046,
\ExplSyntaxOn	6	1054, 1061, 1070, 1076, 1082, 1089,
		1095, 1101, 1107, 1113, 1129, 1135
F		
fp commands:		
\fp_eval:n	368	\msg_show:nnnnn 960
\l_tmpa_fp	5	\msg_show:nnnnnn 949
		\msg_show_item_unbraced:nn 953, 965
I		
\IfInstanceExistsF	1193, 8	muskip commands:
\IfInstanceExistsT	1193, 8	
\IfInstanceExistsTF	1193, 8	
\IncludeInRelease	1219	
int commands:		
\int_compare:nNnTF	45	\muskip_eval:n 366
\int_compare:nTF	191	\muskip_new:N 35
\int_eval:n	362	\l_tmpa_muskip 5
\int_new:N	34	
\int_set:Nn	190	
\l_tmpa_int	5	
\item	6	
K		
kernel internal commands:		
__kernel_quark_new_conditional:Nn	41	
keys commands:		
\keys_define:nn	512, 533, 594	
\keys_set:nn	1215	
\keys_set_known:nnN	1208	
keyval commands:		
\keyval_parse:Nn	212, 617	
\keyval_parse:nnn	413, 534	
\KeyValue	881, 1086, 1199, 79, 4	
M		
\message	3	
msg commands:		
\msg_error:nn	267	
\msg_error:nnn	890,	
	62, 69, 185, 233, 247, 291, 313,	
	420, 433, 450, 506, 544, 623, 632, 738	
\msg_error:nnnn	907, 913,	
	56, 75, 199, 562, 590, 602, 691, 703, 803	
\msg_error:nnnnn	48, 417	
\msg_info:nnnn		
	109, 193, 403, 648, 671, 687	
\msg_line_context:		
	1013, 1019, 1026, 1033, 1040,	
	1047, 1055, 1142, 1144, 1148, 1151	
\g_msg_module_type_prop	1162	
\msg_new:nnn		
	1141, 1143, 1145, 1150, 1152, 1157	
\msg_new:nnnn		
	969, 977, 983, 992, 998, 1005,	

R

regex commands:

- \regex_match:nnTF 385

S

scan commands:

- \scan_new:N 38, 39

scan internal commands:

- \s__template_mark 888, 38, 12
- \s__template_stop 888, 892, 39, 286, 297, 308, 329, 343, 351, 437, 440, 747, 750, 12

\section 2

seq commands:

- \seq_clear:N 160, 211
- \seq_const_from_clist:Nn 16
- \seq_gclear_new:N 115
- \seq_gset_eq:NN 116
- \seq_if_exist:NTF 155
- \seq_if_in:NnTF 230
- \seq_map_break: 249, 340
- \seq_map_function:NN .. 226, 346, 714
- \seq_new:N 29
- \seq_put_right:Nn 263
- \seq_set_eq:NN 157
- \SetKnownTemplateKeys 1201, 6
- \SetTemplateKeys 1201, 6
- \ShowInstanceValues 1183, 10
- \ShowTemplateCode 1183, 10
- \ShowTemplateDefaults 1183, 10
- \ShowTemplateInterface 1183, 10
- \ShowTemplateVariables 1183, 10

skip commands:

- \skip_eval:n 370
- \skip_new:N 36
- \l_tmpa_skip 5

str commands:

- \str_case:nn 517
- \str_case:nnTF 455
- \str_if_eq:nnTF 905, 79, 92, 243, 264, 469, 484, 500, 572, 729, 731
- \str_if_in:nnTF 383

T

template internal commands:

- __template_assign_boolean: 766, 766
- __template_assign_boolean_aux:n 766, 769, 770, 772
- __template_assign_choice: . 792, 792
- __template_assign_choice_-aux:nTF 792, 794, 797, 809, 814
- __template_assign_function: 815, 815
- __template_assign_function_-aux:N 815, 818, 819, 821
- __template_assign_instance: 832, 832
- __template_assign_instance_-aux:N 832, 835, 836, 838
- __template_assign_variable: ... 850, 736, 850
- __template_assign_variable:n ... 852, 860, 850
- __template_assignments_pop: ... 915, 915, 1200
- __template_assignments_push:n .. 916, 916, 674
- \l__template_assignments_tl 865, 872, 915, 917, 19, 675, 713, 777, 785, 812, 823, 840, 11
- \c__template_code_root_tl 9, 919, 54, 402, 405, 647, 649, 650, 676, 763, 11
- __template_convert_to_assignments: 669, 711, 711, 762
- __template_convert_to_assignments_-aux:n 711, 715, 717
- __template_convert_to_assignments_-aux:nn 711, 720, 722, 741
- __template_declare_instance:nnnn 1176, 652, 652
- __template_declare_instance_-aux:nnnn 652, 658, 661, 708
- __template_declare_template_-code:nnnn .. 374, 386, 388, 393, 400
- __template_declare_template_-code:nnnnn 1168, 374, 374
- __template_declare_template_-keys:nnnn 1166, 203, 203
- __template_declare_type:nn 182, 186, 188
- \l__template_default_tl 20, 11
- \c__template_defaults_root_tl ... 10, 99, 100, 139, 142, 11
- __template_define_type:nn 1164, 182, 182
- __template_edit_defaults:nnn ... 1172, 605, 605
- __template_edit_instance:nnn ... 1180, 693, 693
- __template_edit_instance_-aux:nnnn 700, 705, 710
- __template_edit_instance_-aux:nnnnn 693
- \l__template_error_bool . 21, 222, 228, 248, 277, 290, 622, 663, 665, 11
- __template_execute_if_arg_-agree:nnTF 42, 42, 207, 378
- __template_execute_if_code_-exist:nnTF .. 940, 52, 52, 654, 757

```

__template_execute_if_keys_-
  exist:nnTF ..... 71
__template_execute_if_keytype_-
  exist:nTF ..... 58, 58, 64, 224
__template_execute_if_type_-
  exist:nTF ..... 65, 65, 205, 376
__template_find_global: 732, 742, 742
__template_find_global_aux:w ...
  ..... 742, 747, 750
l__template_global_bool .....
  .. 856, 22, 744, 753, 768, 817, 834, 11
__template_if_instance_exist:nn 84
__template_if_instance_exist:nnTF
  911, 957, 1194, 1196, 1198, 84, 682, 695
__template_if_key_value:n .. 77, 83
__template_if_key_value:nTF ...
  ..... 862, 77, 774
__template_if_keys_exist:nnTF ..
  ..... 922, 931, 71, 380, 607
__template_if_use_template:n ... 90
__template_if_use_template:nTF .
  ..... 899, 90
__template_implement_choice_-
  elt:n ..... 535, 568, 600
__template_implement_choice_-
  elt:nnn ..... 536, 568, 568
__template_implement_choice_-
  elt_aux:n ..... 568, 574, 582, 585
__template_implement_choice_-
  elt_aux:nnn ..... 568, 573, 580, 592
__template_implement_choices:nn
  ..... 457, 529, 529
__template_implement_choices_-
  default: ..... 529, 540, 547
__template_instance_set_eq:nnn .
  ..... 1178, 680, 680
c__template_instances_root_tl ..
  11, 912, 86, 670, 672, 686, 688, 689, 11
l__template_key_name_tl .....
  .... 23, 231, 234, 261, 263, 284,
  299, 306, 311, 353, 355, 360, 423,
  426, 431, 475, 489, 504, 513, 532,
  533, 538, 550, 554, 557, 560, 563,
  564, 587, 590, 595, 597, 603, 627,
  629, 632, 734, 795, 798, 800, 804, 12
c__template_key_order_root_tl ..
  ..... 13, 115, 116, 155, 158, 11
l__template_key_order_seq .. 29,
  117, 157, 160, 211, 230, 263, 714, 12
__template_key_to_value: .....
  ..... 864, 879, 879, 776
__template_key_to_value_auxi:w .
  ..... 879, 880, 881
__template_key_to_value_auxii:w
  ..... 879, 887, 892
l__template_keytype_arg_tl ....
  ..... 25, 245, 258, 259,
  266, 323, 337, 470, 531, 828, 845, 12
l__template_keytype_tl . 24, 224,
  243, 257, 264, 273, 322, 333, 427,
  429, 455, 517, 637, 729, 731, 735, 12
c__template_keytypes_arg_seq ...
  ..... 16, 226, 346, 11
l__template_keytypes_prop . 934,
  28, 114, 151, 154, 210, 261, 416,
  425, 431, 557, 587, 629, 719, 800, 12
c__template_keytypes_root_tl 12,
  73, 107, 110, 112, 113, 149, 152, 11
__template_map_var_type: .....
  ..... 854, 496, 499, 515, 515
__template_parse_keys_elt:n ...
  ..... 213, 219, 219, 272, 18
__template_parse_keys_elt:nn ...
  ..... 213, 270, 270
__template_parse_keys_elt_aux: .
  ..... 219, 236, 253
__template_parse_keys_elt_aux:n
  ..... 219, 227, 241
__template_parse_values:nnn ...
  ..... 610, 614, 614, 664, 761
__template_parse_values_elt:n ..
  ..... 618, 620, 620
__template_parse_values_elt:nn .
  ..... 618, 625, 625
__template_parse_values_elt_-
  aux:n ..... 625, 631, 634
__template_parse_vars_elt:n ...
  ..... 414, 419, 419
__template_parse_vars_elt:nnn ..
  ..... 414, 421, 421
__template_parse_vars_elt_-
  aux:nn ..... 421, 430, 435
__template_parse_vars_elt_-
  aux:nnn .... 421, 443, 447, 453, 509
__template_parse_vars_elt_-
  aux:nw ..... 421, 437, 439
__template_parse_vars_elt_-
  key:nn ..... 421, 462, 479, 497, 510
__template_quark_if_nil:N ..... 41
__template_quark_if_nil:NTF .. 894
__template_quark_if_nil:nTF ... 41
__template_quark_if_nil_p:n ... 41
__template_recover_defaults:nn .
  924, 136, 136, 410, 609, 641, 656, 759
__template_recover_keytypes:nn .
  ..... 933, 136, 146, 411, 616, 643

```

__template_recover_values:nn ...	__template_store_value_real:n ..
..... 959, 136, 162, 684, 697 359, 367
__template_recover_vars:nn	__template_store_value_skip:n ..
..... 942, 136, 172, 645, 657, 707, 760 359, 369
\c_template_restrict_root_tl ... 11	__template_store_value_tokenlist:n
__template_show:Nnnn 359, 371, 373
..... 920, 925, 934, 943, 947	__template_store_values:nn
__template_show_code:nn 96, 120, 668, 685
..... 918, 918, 1184	__template_store_vars:nn
__template_show_defaults:nn 96, 128, 415, 646
..... 920, 920, 1186	__template_template_set_eq:nnn .
__template_show_keytypes:nn 1170, 639, 639
..... 920, 929, 1188	\l__template_tmp_clist
__template_show_values:nn 32, 531, 541, 543, 570, 577, 579, 12
..... 955, 955, 1192	\l__template_tmp_dim
__template_show_vars:nn 33, 12
..... 920, 938, 1190	\l__template_tmp_int
__template_split_keytype:n 34, 190, 191, 194, 196, 200, 12
..... 221, 275, 275	\l__template_tmp_muskip
__template_split_keytype_arg:n 35, 12
..... 316, 320,	\l__template_tmp_skip
320, 349, 429, 559, 589, 636, 728, 802 36, 12
__template_split_keytype_arg_-	\l__template_tmp_tl
aux:n	884, 890, 37, 44, 45, 49, 255, 262,
320, 324, 347, 350	278, 279, 280, 286, 539, 549, 550,
__template_split_keytype_arg_-	551, 553, 554, 555, 558, 559, 561,
aux:w	588, 589, 596, 598, 630, 636, 699,
320, 328, 343, 351	701, 719, 720, 801, 802, 811, 812, 12
__template_split_keytype_aux:w .	\g__template_type_prop
..... 275, 285, 296, 308 18, 44, 67, 184, 195, 11
__template_store_defaults:nn ...	__template_use_instance:nn
..... 96, 96, 214, 611, 642 897, 897, 1182, 844
__template_store_key_implementation:nnn	__template_use_instance_aux:nn .
..... 382, 408, 408 897, 901, 909
__template_store_keytypes:nn ...	__template_use_instance_-
..... 96, 104, 215, 644	aux:nNnnn
__template_store_value:n	897, 900, 903
..... 354, 354, 356, 357, 358	__template_use_template:nnn ...
__template_store_value_aux:Nn 906, 1174, 755, 755
359, 359, 362, 364, 366, 368, 370, 372	\l__template_value_tl .. 862, 868,
__template_store_value_boolean:n	875, 880, 885, 887, 895, 26, 724,
..... 352, 352	774, 781, 787, 795, 805, 829, 846, 12
__template_store_value_choice:n	\l__template_values_prop
..... 354, 356 925, 964, 30, 101,
__template_store_value_commalist:n	125, 141, 144, 167, 170, 209, 353,
..... 359, 373	355, 360, 538, 560, 667, 698, 724, 21
__template_store_value_function:n	\c__template_values_root_tl
..... 354, 357 14, 123, 124, 165, 168, 11
__template_store_value_instance:n	\l__template_var_tl
..... 354, 358 867, 874, 27, 726,
__template_store_value_integer:n	745, 747, 752, 780, 788, 826, 842, 12
..... 359, 361	\l__template_vars_prop .. 884, 943,
__template_store_value_length:n	31, 133, 177, 180, 412, 474, 488,
..... 359, 363	503, 532, 551, 555, 598, 726, 811, 12
__template_store_value_muskip:n	\c__template_vars_root_tl
..... 359, 365 15, 131, 132, 175, 178, 11
	tl commands:
	\c_empty_tl
	1205

```

\c_space_tl ... 550, 554, 597, 795, 798
\tl_clear:N ... 284, 323, 713
\tl_const:Nn .. 9, 10, 11, 12, 13, 14, 15
\tl_head:w ... 79, 92
\tl_if_blank:nTF ... 331, 335, 442, 445
\tl_if_empty:NTF ... 245, 258, 311
\tl_if_empty:nTF 1155, 1160, 1203, 1213
\tl_if_in:nnTF ... 280, 304, 326, 745
\tl_if_single:nTF ... 493
\tl_new:N ...
... 19, 20, 23, 24, 25, 26, 27, 37, 1218
\tl_put_right:Nn ... 865,
872, 299, 306, 777, 785, 812, 823, 840
\tl_replace_all:Nnn ... 279
\tl_set:Nn ...
. 883, 895, 917, 255, 278, 322, 333,
337, 423, 549, 553, 596, 627, 734, 752

\tl_set_eq:NN ... 1205
\tl_to_str:n ... 883,
950, 951, 952, 961, 962, 302, 424, 628
\tl_trim_spaces:n ...
... 883, 301, 322, 334, 424, 448, 628
\l_tmpa_tl ... 5
token commands:
\token_to_str:N ...
... 1086, 279, 280, 297, 304, 307, 314

U
\UnusedTemplateKeys .. 1205, 1208, 1218, 6
use commands:
\use:N ... 912, 273, 496, 637, 763
\use:n ... 294, 338, 372
\use_i:nn ... 5
\UseInstance ... 1079, 1163, 485, 9
\UseTemplate ... 1163, 92, 9

```