

Templates: Prototype document functions^{*}

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

September 15, 2026

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 standardize 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 v1.0l dated 2026-08-14, © 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 $\text{\TeX}$ 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 defines 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 (protected) 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 `choice` and `function` all of these accept the keyword `global` to carry out a global assignment. Apart from `choice`, all of these accept the keyword `name` to allow passing command name without the leading backslash: this is useful for auto-generated names.

<code>\DeclareTemplateCode</code>	<code>\DeclareTemplateCode</code>
	<code>{⟨type⟩} {⟨template⟩} {⟨no. of args⟩}</code>
	<code>{⟨key bindings⟩} {⟨code⟩}</code>

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

```

⟨key1⟩ = ⟨variable1⟩,
⟨key2⟩ = ⟨variable2⟩,
⟨key3⟩ = global ⟨variable3⟩,
⟨key4⟩ = global ⟨variable4⟩,
⟨key5⟩ = name { ⟨name5⟩ },
...
```

As illustrated, the keyword `global` may be included in the listing to indicate that the `⟨variable⟩` should be assigned globally. The keyword `name` may also be used to give the variable by (cs)name rather than as a single token. If both keywords are used, `global` should come before `name`.

A full list of variable bindings is given in Table 2.

The `⟨code⟩` 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 `⟨number of arguments⟩` 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

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.

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

<code>\DeclareTemplateCopy</code>	<code>\DeclareTemplateCopy</code> <code>{<type>} {<template2>} {<template1>}</code>
-----------------------------------	--

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.

```

\DeclareInstance \DeclareInstance
                 {\type} {\instance} {\template} {\parameters}

```

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

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.

\InstanceValue *	\InstanceValue {<type>} {<instance>} {<key>}
	Expands to the current value for the <key> stored in the <instance> of <type>. If the <instance> or <key> do not exist, the expansion is empty. The result is returned within the \unexpanded primitive (\exp_not:n).
\IfInstanceExistsT \IfInstanceExistsF \IfInstanceExistsTF	\IfInstanceExistsTF {<type>} {<instance>} {<true code>} {<false code>}
	Tests if the named <instance> of a <type> exists, and then inserts the appropriate code into the input stream.
\DeclareInstanceCopy	\DeclareInstanceCopy {<type>} {<instance2>} {<instance1>}
	Copies the <values> for <instance1> for an <type> to <instance2>.

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.

\UseInstance	\UseInstance {<type>} {<instance>} <arguments>
	Uses an <instance> of the <type>, which will require <arguments> as determined by the number specified for the <type>. The <instance> must have been declared before it can be used, otherwise an error is raised.
\UseTemplate	\UseTemplate {<type>} {<template>} {<settings>} <arguments>
	Uses the <template> of the specified <type>, applying the <settings> and absorbing <arguments> as detailed by the <type> declaration. This in effect is the same as creating an instance using \DeclareInstance and immediately using it with \UseInstance , 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.

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

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

9.1 Expanding the values of keys

To allow the user to apply expansion of values when the key is set, key names can be followed by an expansion specifier. This is given by appending `:` and a single letter specifier to the key name. These letters are the normal argument specifiers for `expl3`, thus they may be one of `n` (redundant but supported), `o`, `V`, `v`, `e`, `N` (again redundant) or `c`. Expansion of a control sequence name is particularly useful when you need to refer to an internal $\text{\LaTeX} 2_{\epsilon}$ or an L3 programming layer variable, e.g.,

```
key-a:c = @itemdepth , % use \@itemdepth as the value
key-b:v = @itemdepth   % use the current value of \@itemdepth as the value
```

10 Getting information about templates and instances

<code>\ShowInstanceValues</code>	<code>\ShowInstanceValues {⟨type⟩} {⟨instance⟩}</code> Shows the <code>⟨values⟩</code> for an <code>⟨instance⟩</code> of the given <code>⟨type⟩</code> at the terminal.
----------------------------------	--

<code>\ShowTemplateCode</code>	<code>\ShowTemplateCode {⟨type⟩} {⟨template⟩}</code> Shows the <code>⟨code⟩</code> of a <code>⟨template⟩</code> for an <code>⟨type⟩</code> in the terminal.
--------------------------------	--

<code>\ShowTemplateDefaults</code>	<code>\ShowTemplateDefaults {⟨type⟩} {⟨template⟩}</code> Shows the <code>⟨default⟩</code> values of a <code>⟨template⟩</code> for an <code>⟨type⟩</code> in the terminal.
------------------------------------	--

<code>\ShowTemplateInterface</code>	<code>\ShowTemplateInterface {⟨type⟩} {⟨template⟩}</code> Shows the <code>⟨keys⟩</code> and associated <code>⟨key types⟩</code> of a <code>⟨template⟩</code> for an <code>⟨type⟩</code> in the terminal.
-------------------------------------	---

<code>\ShowTemplateVariables</code>	<code>\ShowTemplateVariables {<type>} {<template>}</code>
-------------------------------------	---

Shows the `<variables>` and associated `<keys>` of a `<template>` for an `<type>` in the terminal. Note that `code` and `choice` keys do not map directly to variables but to arbitrary code. For `choice` 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 Debugging support

<code>\DebugTemplatesOn</code>	<code>\DebugTemplatesOn</code>
<code>\DebugTemplatesOff</code>	

Turn on or off debugging support for use of templates. Debugging information is provided at the point of *use* of templates and instances, i.e. where normal messages should be avoided for performance reasons.

12 The implementation

```

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

```

12.1 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_seq

A list of all keytypes.

```

16 \exp_args:Nne \seq_const_from_clist:Nn \c__template_keytypes_seq
17 {
18   \tl_to_str:n
19   {
20     boolean   ,
21     choice    ,
22     commalist ,
23     function  ,
24     instance  ,
25     integer   ,
26     length    ,
27     muskip    ,
28     real      ,
29     skip      ,
30     tokenlist
31   }
32 }

```

\c__template_keytypes_arg_seq

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

```

33 \seq_const_from_clist:Nn \c__template_keytypes_arg_seq
34 { choice , function , instance }

```

\g__template_type_prop

For storing types and the associated number of arguments.

```

35 \prop_new:N \g__template_type_prop

```

\l__template_assignments_tl

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

36 \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.

37 \tl_new:N \l__template_default_tl

\l__template_error_bool

A flag for errors to be carried forward.

38 \bool_new:N \l__template_error_bool

\l__template_global_bool

Used to indicate that assignments should be global.

39 \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.

40 \tl_new:N \l__template_key_name_tl

41 \tl_new:N \l__template_keytype_tl

42 \tl_new:N \l__template_keytype_arg_tl

43 \tl_new:N \l__template_value_tl

44 \tl_new:N \l__template_var_tl

\l__template_value_exp_str

45 \str_new:N \l__template_value_exp_str

\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

46 \prop_new:N \l__template_keytypes_prop

47 \seq_new:N \l__template_key_order_seq

48 \prop_new:N \l__template_values_prop

49 \prop_new:N \l__template_vars_prop

<code>\l__template_tmp_clist</code>	Scratch space.
<code>\l__template_tmp_dim</code>	
<code>\l__template_tmp_int</code>	50 <code>\clist_new:N \l__template_tmp_clist</code>
<code>\l__template_tmp_muskip</code>	51 <code>\dim_new:N \l__template_tmp_dim</code>
<code>\l__template_tmp_skip</code>	52 <code>\int_new:N \l__template_tmp_int</code>
<code>\l__template_tmp_tl</code>	53 <code>\muskip_new:N \l__template_tmp_muskip</code>
	54 <code>\skip_new:N \l__template_tmp_skip</code>
	55 <code>\tl_new:N \l__template_tmp_tl</code>

<code>\s__template_mark</code>	Internal scan marks.
<code>\s__template_stop</code>	
	56 <code>\scan_new:N \s__template_mark</code>
	57 <code>\scan_new:N \s__template_stop</code>

<code>\q__template_nil</code>	Internal quarks.
	58 <code>\quark_new:N \q__template_nil</code>

<code>__template_quark_if_nil_p:n</code>	Branching quark conditional.
<code>__template_quark_if_nil:nTF</code>	59 <code>__kernel_quark_new_conditional:Nn __template_quark_if_nil:N { F }</code>

(End of definition for `\__template_quark_if_nil:nTF`.)

12.2 Debugging support

<code>__template_debug_on:</code>	
<code>__template_debug_off:</code>	60 <code>\cs_new_protected:Npn __template_debug_on:</code>
	61 <code>{</code>
	62 <code>\cs_set_protected:Npn __template_debug:n ##1 {##1}</code>
	63 <code>\cs_set_protected:Npn __template_debug_typeout:n ##1</code>
	64 <code>{ \iow_term:e { [Template] ~ ==> ~ ##1 } }</code>
	65 <code>}</code>
	66 <code>\cs_new_protected:Npn __template_debug_off:</code>
	67 <code>{</code>
	68 <code>\cs_set_protected:Npn __template_debug:n ##1 { }</code>
	69 <code>\cs_set_protected:Npn __template_debug_typeout:n ##1 { }</code>
	70 <code>}</code>

(End of definition for `\__template_debug_on:` and `\__template_debug_off:.`)

<code>__template_debug:n</code>	
<code>__template_debug_typeout:n</code>	71 <code>\cs_new_protected:Npn __template_debug:n #1 { }</code>
	72 <code>\cs_new_protected:Npn __template_debug_typeout:n #1 { }</code>

(End of definition for `\__template_debug:n` and `\__template_debug_typeout:n`.)

12.3 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

```

73 \cs_new_protected:Npn \__template_execute_if_arg_agree:nnT #1#2#3
74 {
75   \prop_get:NnN \g__template_type_prop {#1} \l__template_tmp_tl
76   \int_compare:nNnTF {#2} = \l__template_tmp_tl
77     {#3}
78   {
79     \msg_error:nneee { template } { argument-number-mismatch }
80     {#1} { \l__template_tmp_tl } {#2}
81   }
82 }
```

(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.

```

83 \cs_new_protected:Npn \__template_execute_if_code_exist:nnT #1#2#3
84 {
85   \cs_if_exist:cTF { \c__template_code_root_tl #1 / #2 }
86     {#3}
87     { \msg_error:nnnn { template } { no-template-code } {#1} {#2} }
88 }
```

(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.

```

89 \cs_new_protected:Npn \__template_execute_if_keytype_exist:nT #1#2
90 {
91   \seq_if_in:NeTF \c__template_keytypes_seq { \tl_to_str:n {#1} }
92     {#2}
93     { \msg_error:nnn { template } { unknown-keytype } {#1} }
94 }
95 \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.

```

96 \cs_new_protected:Npn \__template_execute_if_type_exist:nT #1#2
97 {
98   \prop_if_in:NnTF \g__template_type_prop {#1}
99     {#2}
100   { \msg_error:nnn { template } { unknown-type } {#1} }
101 }
```

(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.

```

102 \cs_new_protected:Npn \_template\_if\_keys\_exist:nnT #1#2#3
103 {
104   \cs\_if\_exist:cTF { \c\_template\_keytypes\_root\_tl #1 / #2 }
105     {#3}
106     { \msg\_error:nnnn { template } { unknown-template } {#1} {#2} }
107 }

```

(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`

```

108 \prg\_new\_conditional:Npnn \_template\_if\_key\_value:n #1 { T , F , TF }
109 {
110   \str\_if\_eq:noTF { \KeyValue } { \tl\_head:w #1 \q\_nil \q\_stop }
111   \prg\_return\_true:
112   \prg\_return\_false:
113 }
114 \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

```

115 \prg\_new\_conditional:Npnn \_template\_if\_instance\_exist:nn #1#2 { T, F, TF }
116 {
117   \cs\_if\_exist:cTF { \c\_template\_instances\_root\_tl #1 / #2 }
118   \prg\_return\_true:
119   \prg\_return\_false:
120 }

```

(End of definition for _template_if_instance_exist:nnTF.)

`\_template\_if\_use\_template:nTF` Tests for the first token in a string being `\UseTemplate`.

```

121 \prg\_new\_conditional:Npnn \_template\_if\_use\_template:n #1 { TF }
122 {
123   \str\_if\_eq:noTF { \UseTemplate } { \tl\_head:w #1 \q\_nil \q\_stop }
124   \prg\_return\_true:
125   \prg\_return\_false:
126 }

```

(End of definition for _template_if_use_template:nTF.)

12.4 Saving and recovering property lists

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

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

`\_template\_store\_defaults:nn`

`\_template\_store\_keytypes:nn`

`\_template\_store\_values:nn`

`\_template\_store\_vars:nn`

```

127 \cs\_new\_protected:Npn \_template\_store\_defaults:nn #1#2
128 {
129   \debug\_suspend:
130   \prop\_gclear\_new:c { \c\_template\_defaults\_root\_tl #1 / #2 }
131   \prop\_gset\_eq:cN { \c\_template\_defaults\_root\_tl #1 / #2 }
132   \l\_template\_values\_prop

```

```

133     \debug_resume:
134 }
135 \cs_new_protected:Npn \__template_store_keytypes:nn #1#2
136 {
137     \debug_suspend:
138     \prop_if_exist:cTF { \c__template_keytypes_root_tl #1 / #2 }
139     {
140         \msg_info:nnnn { template } { declare-template-interface } {#1} {#2}
141         \prop_gclear:c { \c__template_keytypes_root_tl #1 / #2 }
142     }
143     { \prop_new:c { \c__template_keytypes_root_tl #1 / #2 } }
144     \prop_gset_eq:cN { \c__template_keytypes_root_tl #1 / #2 }
145     \l__template_keytypes_prop
146     \seq_gclear_new:c { \c__template_key_order_root_tl #1 / #2 }
147     \seq_gset_eq:cN { \c__template_key_order_root_tl #1 / #2 }
148     \l__template_key_order_seq
149     \debug_resume:
150 }
151 \cs_new_protected:Npn \__template_store_values:nn #1#2
152 {
153     \debug_suspend:
154     \prop_clear_new:c { \c__template_values_root_tl #1 / #2 }
155     \prop_set_eq:cN { \c__template_values_root_tl #1 / #2 }
156     \l__template_values_prop
157     \debug_resume:
158 }
159 \cs_new_protected:Npn \__template_store_vars:nn #1#2
160 {
161     \debug_suspend:
162     \prop_gclear_new:c { \c__template_vars_root_tl #1 / #2 }
163     \prop_gset_eq:cN { \c__template_vars_root_tl #1 / #2 }
164     \l__template_vars_prop
165     \debug_resume:
166 }

```

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

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_defaults:nn
\__template_recover_keytypes:nn
\__template_recover_values:nn
\__template_recover_vars:nn
167 \cs_new_protected:Npn \__template_recover_defaults:nn #1#2
168 {
169     \prop_if_exist:cTF
170     { \c__template_defaults_root_tl #1 / #2 }
171     {
172         \prop_set_eq:Nc \l__template_values_prop
173         { \c__template_defaults_root_tl #1 / #2 }
174     }
175     { \prop_clear:N \l__template_values_prop }
176 }
177 \cs_new_protected:Npn \__template_recover_keytypes:nn #1#2
178 {
179     \prop_if_exist:cTF
180     { \c__template_keytypes_root_tl #1 / #2 }

```

```

181     {
182         \prop_set_eq:Nc \l__template_keytypes_prop
183         { \c__template_keytypes_root_tl #1 / #2 }
184     }
185     { \prop_clear:N \l__template_keytypes_prop }
186 \seq_if_exist:cTF { \c__template_key_order_root_tl #1 / #2 }
187     {
188         \seq_set_eq:Nc \l__template_key_order_seq
189         { \c__template_key_order_root_tl #1 / #2 }
190     }
191     { \seq_clear:N \l__template_key_order_seq }
192 }
193 \cs_new_protected:Npn \__template_recover_values:nn #1#2
194 {
195     \prop_if_exist:cTF
196     { \c__template_values_root_tl #1 / #2 }
197     {
198         \prop_set_eq:Nc \l__template_values_prop
199         { \c__template_values_root_tl #1 / #2 }
200     }
201     { \prop_clear:N \l__template_values_prop }
202 }
203 \cs_new_protected:Npn \__template_recover_vars:nn #1#2
204 {
205     \prop_if_exist:cTF
206     { \c__template_vars_root_tl #1 / #2 }
207     {
208         \prop_set_eq:Nc \l__template_vars_prop
209         { \c__template_vars_root_tl #1 / #2 }
210     }
211     { \prop_clear:N \l__template_vars_prop }
212 }

```

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

12.5 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.

```

213 \cs_new_protected:Npn \__template_define_type:nn #1#2
214 {
215     \prop_if_in:NnTF \g__template_type_prop {#1}
216     { \msg_error:nnn { template } { type-already-defined } {#1} }
217     { \__template_declare_type:nn {#1} {#2} }
218 }
219 \cs_new_protected:Npn \__template_declare_type:nn #1#2
220 {
221     \int_set:Nn \l__template_tmp_int {#2}
222     \int_compare:nTF { 0 <= \l__template_tmp_int <= 9 }
223     {
224         \msg_info:nnnV { template } { declare-type }
225         {#1} \l__template_tmp_int
226         \prop_gput:NnV \g__template_type_prop {#1}

```

```

227         \l__template_tmp_int
228     }
229     {
230         \msg_error:nnnV { template } { bad-number-of-arguments }
231         {#1} \l__template_tmp_int
232     }
233 }

```

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

12.6 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 initialized. The mechanism is then set up for the l3keys module to actually parse the keys. Finally, the code hands off to the storage routine to save the parsed information properly.

```

234 \cs_new_protected:Npn \__template_declare_template_keys:nnnn #1#2#3#4
235 {
236     \__template_execute_if_type_exist:nT {#1}
237     {
238         \__template_execute_if_arg_agree:nnT {#1} {#3}
239         {
240             \prop_clear:N \l__template_values_prop
241             \prop_clear:N \l__template_keytypes_prop
242             \seq_clear:N \l__template_key_order_seq
243             \keyval_parse:NNn
244             \__template_parse_keys_elt:n \__template_parse_keys_elt:nn {#4}
245             \__template_store_defaults:nn {#1} {#2}
246             \__template_store_keytypes:nn {#1} {#2}
247         }
248     }
249 }

```

(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.

```

250 \cs_new_protected:Npn \__template_parse_keys_elt:n #1
251 {
252     \__template_split_keytype:n {#1}
253     \bool_if:NF \l__template_error_bool
254     {
255         \__template_execute_if_keytype_exist:VT \l__template_keytype_tl

```

```

256     {
257         \seq_map_function:NN \c__template_keytypes_arg_seq
258         \__template_parse_keys_elt_aux:n
259         \bool_if:NF \l__template_error_bool
260         {
261             \seq_if_in:NoTF \l__template_key_order_seq
262             \l__template_key_name_tl
263             {
264                 \msg_error:nnV { template } { duplicate-key-interface }
265                 \l__template_key_name_tl
266             }
267             { \__template_parse_keys_elt_aux: }
268         }
269     }
270 }
271 }
272 \cs_new_protected:Npn \__template_parse_keys_elt_aux:n #1
273 {
274     \str_if_eq:VnT \l__template_keytype_tl {#1}
275     {
276         \tl_if_empty:NT \l__template_keytype_arg_tl
277         {
278             \msg_error:nnn { template } { keytype-requires-argument } {#1}
279             \bool_set_true:N \l__template_error_bool
280             \seq_map_break:
281         }
282     }
283 }
284 \cs_new_protected:Npn \__template_parse_keys_elt_aux:
285 {
286     \tl_set:Ne \l__template_tmp_tl
287     {
288         \l__template_keytype_tl
289         \tl_if_empty:NF \l__template_keytype_arg_tl
290         { { \l__template_keytype_arg_tl } }
291     }
292     \prop_put:NVV \l__template_keytypes_prop \l__template_key_name_tl
293     \l__template_tmp_tl
294     \seq_put_right:NV \l__template_key_order_seq \l__template_key_name_tl
295     \str_if_eq:VnT \l__template_keytype_tl { choice }
296     {
297         \clist_if_in:NnT \l__template_keytype_arg_tl { unknown }
298         { \msg_error:nn { template } { choice-unknown-reserved } }
299     }
300 }

```

(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.

```

301 \cs_new_protected:Npn \__template_parse_keys_elt:nn #1#2
302 {

```

```

303     \__template_parse_keys_elt:n {#1}
304     \prop_put:Non \l__template_values_prop \l__template_key_name_tl {#2}
305 }

```

(End of definition for __template_parse_keys_elt:nn.)

__template_split_keytype:n 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 standardized. After that, the standard delimited argument method is used to separate the two parts.

```

306 \cs_new_protected:Npe \__template_split_keytype:n #1
307 {
308     \exp_not:N \bool_set_false:N \exp_not:N \l__template_error_bool
309     \tl_set:Nn \exp_not:N \l__template_tmp_tl {#1}
310     \tl_replace_all:Nnn \exp_not:N \l__template_tmp_tl { : } { \token_to_str:N : }
311     \tl_if_in:NnTF \exp_not:N \l__template_tmp_tl { \token_to_str:N : }
312     {
313         \exp_not:n
314         {
315             \tl_clear:N \l__template_key_name_tl
316             \exp_after:wN \__template_split_keytype_aux:w
317             \l__template_tmp_tl \s__template_stop
318         }
319     }
320     {
321         \exp_not:N \bool_set_true:N \exp_not:N \l__template_error_bool
322         \msg_error:nnn { template } { missing-keytype } {#1}
323     }
324 }
325 \use:e
326 {
327     \cs_new_protected:Npn \exp_not:N \__template_split_keytype_aux:w
328     #1 \token_to_str:N : #2 \s__template_stop
329     {
330         \tl_put_right:Ne \exp_not:N \l__template_key_name_tl
331         {
332             \exp_not:N \tl_trim_spaces:e
333             { \exp_not:N \tl_to_str:n {#1} }
334         }
335         \tl_if_in:nnTF {#2} { \token_to_str:N : }
336         {
337             \tl_put_right:Nn \exp_not:N \l__template_key_name_tl
338             { \token_to_str:N : }
339             \exp_not:N \__template_split_keytype_aux:w #2 \s__template_stop
340         }
341         {
342             \exp_not:N \tl_if_empty:NTF \exp_not:N \l__template_key_name_tl
343             {
344                 \msg_error:nnn { template } { empty-key-name }
345                 { \token_to_str:N : #2 }
346             }
347             { \exp_not:N \__template_split_keytype_arg:n {#2} }
348         }
349     }
350 }

```

(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).

```

351 \cs_new_protected:Npn __template_split_keytype_arg:n #1
352 {
353   \tl_set:Nc \l__template_keytype_tl { \tl_trim_spaces:n {#1} }
354   \tl_clear:N \l__template_keytype_arg_tl
355   \cs_set_protected:Npn __template_split_keytype_arg_aux:n ##1
356   {
357     \tl_if_in:nnT {#1} {##1}
358     {
359       \cs_set:Npn __template_split_keytype_arg_aux:w
360       ####1 ##1 ####2 \s__template_stop
361       {
362         \tl_if_blank:nT {####1}
363         {
364           \tl_set:Nc \l__template_keytype_tl
365           { \tl_trim_spaces:n {##1} }
366           \tl_if_blank:nF {####2}
367           {
368             \tl_set:Nc \l__template_keytype_arg_tl
369             { \use:n {####2} }
370           }
371           \seq_map_break:
372         }
373       }
374       __template_split_keytype_arg_aux:w #1 \s__template_stop
375     }
376   }
377   \seq_map_function:NN \c__template_keytypes_arg_seq
378   __template_split_keytype_arg_aux:n
379 }
380 \cs_generate_variant:Nn __template_split_keytype_arg:n { V }
381 \cs_new:Npn __template_split_keytype_arg_aux:n #1 { }
382 \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`.)

12.7 Implementation part of template declaration

```

__template_declare_template_code:nnnnn
__template_declare_template_code:nnnn

```

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.

```

383 \cs_new_protected:Npn __template_declare_template_code:nnnnn #1#2#3#4#5
384 {
385   __template_execute_if_type_exist:nT {#1}

```

```

386 {
387     \__template_execute_if_arg_agree:nnT {#1} {#3}
388     {
389         \__template_if_keys_exist:nnT {#1} {#2}
390         {
391             \__template_store_key_implementation:nnn {#1} {#2} {#4}
392             \str_if_in:nnTF {#5} { AssignTemplateKeys }
393             {
394                 \regex_match:nnTF { \c { AssignTemplateKeys } } {#5}
395                 { \__template_declare_template_code:nnnn {#1} {#2} {#3} {#5} }
396                 {
397                     \__template_declare_template_code:nnnn
398                     {#1} {#2} {#3} { \AssignTemplateKeys #5 }
399                 }
400             }
401             {
402                 \__template_declare_template_code:nnnn
403                 {#1} {#2} {#3} { \AssignTemplateKeys #5 }
404             }
405         }
406     }
407 }
408 }
409 \cs_new_protected:Npn \__template_declare_template_code:nnnn #1#2#3#4
410 {
411     \cs_if_exist:cT { \c__template_code_root_tl #1 / #2 }
412     { \msg_info:nnnn { template } { declare-template-code } {#1} {#2} }
413     \cs_generate_from_arg_count:cNnn
414     { \c__template_code_root_tl #1 / #2 }
415     \cs_gset_protected:Npn {#3} {#4}
416 }

```

(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.

```

417 \cs_new_protected:Npn \__template_store_key_implementation:nnn #1#2#3
418 {
419     \__template_recover_defaults:nn {#1} {#2}
420     \__template_recover_keytypes:nn {#1} {#2}
421     \prop_clear:N \l__template_vars_prop
422     \keyval_parse:nnn
423     { \__template_parse_vars_elt:n } { \__template_parse_vars_elt:nnn { #1 / #2 } } {#3}
424     \__template_store_vars:nn {#1} {#2}
425     \prop_map_inline:Nn \l__template_keytypes_prop
426     { \msg_error:nnnnn { template } { key-not-implemented } {##1} {#2} {#1} }
427 }

```

(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.

```

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

```

(End of definition for _template_parse_vars_elt:n.)

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.

```

\_template_parse_vars_elt:nnn
\_template_parse_vars_elt_auxi:nn
\_template_parse_vars_elt_auxii:nw
\_template_parse_vars_elt_auxiii:nnn
\_template_parse_vars_elt_auxiii:enn
\_template_parse_vars_elt_auxiv:nnw
\_template_parse_vars_elt_auxv:nnn
\_template_parse_vars_elt_auxv:enn
\_template_parse_vars_elt_key:nn
430 \cs_new_protected:Npn \_template_parse_vars_elt:nnn #1#2#3
431 {
432   \tl_set:Nx \l__template_key_name_tl
433     { \tl_trim_spaces:e { \tl_to_str:n {#2} } }
434   \prop_get:NVNTF \l__template_keytypes_prop
435     \l__template_key_name_tl
436     \l__template_keytype_tl
437     {
438       \_template_split_keytype_arg:V \l__template_keytype_tl
439       \_template_parse_vars_elt_auxi:nn {#1} {#3}
440       \prop_remove:NV \l__template_keytypes_prop \l__template_key_name_tl
441     }
442   { \msg_error:nnn { template } { unknown-key } {#2} }
443 }

```

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

```

444 \cs_new_protected:Npn \_template_parse_vars_elt_auxi:nn #1#2
445 {
446   \_template_parse_vars_elt_auxii:nw {#1} #2 global global \s__template_stop
447 }
448 \cs_new_protected:Npn \_template_parse_vars_elt_auxii:nw
449   #1#2 global #3 global #4 \s__template_stop
450 {
451   \tl_if_blank:nTF {#4}
452   { \_template_parse_vars_elt_auxiii:nnn {#2} {#1} { } }
453   {
454     \tl_if_blank:nTF {#2}
455     {
456       \_template_parse_vars_elt_auxiii:enn
457       { \tl_trim_spaces:n {#3} }
458       {#1} { global }
459     }
460     { \msg_error:nnn { template } { bad-variable } { #2 global #3 } }
461   }
462 }
463 \cs_new_protected:Npn \_template_parse_vars_elt_auxiii:nnn #1#2#3
464 {
465   \str_case:VnF \l__template_keytype_tl
466   {
467     { choice } { \_template_implement_choices:nn {#2} {#1} }
468     { function }
469     {
470       \cs_if_exist:NF #1
471       { \cs_new:Npn #1 { } }

```

```

472     \_template_parse_vars_elt_key:nn {#2}
473     {
474         .code:n =
475         {
476             \cs_generate_from_arg_count:NNnn
477             \exp_not:N #1
478             \exp_not:c
479             { cs_ \str_if_eq:nnT {#3} { global } { g } set:Npn }
480             { \exp_not:V \l__template_keytype_arg_tl }
481             {##1}
482         }
483     }
484     \prop_put:NVn \l__template_vars_prop
485     \l__template_key_name_tl {#3#1}
486 }
487 { instance }
488 {
489     \_template_parse_vars_elt_key:nn {#2}
490     {
491         .code:n =
492         {
493             \exp_not:c
494             { cs_ \str_if_eq:nnT {#3} { global } { g } set:Npn }
495             \exp_not:N #1 { \UseInstance {##1} }
496         }
497     }
498     \prop_put:NVn \l__template_vars_prop
499     \l__template_key_name_tl {#3#1}
500 }
501 }
502 {
503     \_template_parse_vars_elt_auxiv:nnw {#2} {#3} #1 name name \s__template_stop
504 }
505 }
506 \cs_generate_variant:Nn \_template_parse_vars_elt_auxiii:nnn { e }
507 \cs_new_protected:Npn \_template_parse_vars_elt_auxiv:nnw
508 #1#2#3 name #4 name #5 \s__template_stop
509 {
510     \tl_if_blank:nTF {#5}
511     { \_template_parse_vars_elt_auxv:nnn {#3} {#1} {#2} }
512     {
513         \tl_if_blank:nTF {#3}
514         {
515             \_template_parse_vars_elt_auxv:enn
516             { \exp_not:c { \tl_trim_spaces:n {#4} } } {#1} {#2}
517         }
518         { \msg_error:nnn { template } { bad-variable } { #3 name #4 } }
519     }
520 }
521 \cs_new_protected:Npn \_template_parse_vars_elt_auxv:nnn #1#2#3
522 {
523     \tl_if_single:nTF {#1}
524     {
525         \cs_if_exist:NF #1

```

```

526         { \use:c { \__template_map_var_type: _new:N } #1 }
527     \__template_parse_vars_elt_key:nn {#2}
528     {
529         . \__template_map_var_type:
530         _ \str_if_eq:nnT {#2} { global } { g } set:N
531         = \exp_not:N #1
532     }
533     \prop_put:NVn \l__template_vars_prop
534     \l__template_key_name_tl {#3#1}
535 }
536 { \msg_error:nnn { template } { bad-variable } {#3#1} }
537 }
538 \cs_generate_variant:Nn \__template_parse_vars_elt_auxv:nnn { e }
539 \cs_new_protected:Npn \__template_parse_vars_elt_key:nn #1#2
540 {
541     \keys_define:ne { template / #1 }
542     { \l__template_key_name_tl #2 }
543 }

```

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

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

```

544 \cs_new:Npn \__template_map_var_type:
545 {
546     \str_case:Vn \l__template_keytype_tl
547     {
548         { boolean } { bool }
549         { commalist } { clist }
550         { integer } { int }
551         { length } { dim }
552         { muskip } { muskip }
553         { real } { fp }
554         { skip } { skip }
555         { tokenlist } { tl }
556     }
557 }

```

(End of definition for __template_map_var_type:.)

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

```

558 \cs_new_protected:Npn \__template_implement_choices:nn #1#2
559 {
560     \clist_set:NV \l__template_tmp_clist \l__template_keytype_arg_tl
561     \prop_put:NVn \l__template_vars_prop \l__template_key_name_tl { }
562     \keys_define:ne { template / #1 } { \l__template_key_name_tl .choice: }
563     \keyval_parse:nnn
564     { \__template_implement_choice_elt:n }
565     { \__template_implement_choice_elt:nnn {#1} }
566     {#2}
567     \prop_get:NVNT \l__template_values_prop \l__template_key_name_tl
568     \l__template_tmp_tl
569     { \__template_implement_choices_default: }
570     \clist_if_empty:NF \l__template_tmp_clist

```

```

571     {
572         \clist_map_inline:Nn \l__template_tmp_clist
573         { \msg_error:nnn { template } { choice-not-implemented } {##1} }
574     }
575 }

```

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

```

576 \cs_new_protected:Npn \__template_implement_choices_default:
577 {
578     \tl_set:Ne \l__template_tmp_tl
579     { \l__template_key_name_tl \c_space_tl \l__template_tmp_tl }
580     \prop_if_in:NVF \l__template_vars_prop \l__template_tmp_tl
581     {
582         \tl_set:Ne \l__template_tmp_tl
583         { \l__template_key_name_tl \c_space_tl \l__template_tmp_tl }
584         \prop_if_in:NVF \l__template_vars_prop \l__template_tmp_tl
585         {
586             \prop_get:NVN \l__template_keytypes_prop \l__template_key_name_tl
587             \l__template_tmp_tl
588             \__template_split_keytype_arg:V \l__template_tmp_tl
589             \prop_get:NVN \l__template_values_prop \l__template_key_name_tl
590             \l__template_tmp_tl
591             \msg_error:nnVV { template } { unknown-default-choice }
592             \l__template_key_name_tl
593             \l__template_key_name_tl
594         }
595     }
596 }

```

(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.

```

597 \cs_new_protected:Npn \__template_implement_choice_elt:nnn #1#2#3
598 {
599     \clist_if_empty:NTF \l__template_tmp_clist
600     {
601         \str_if_eq:nnTF {#2} { unknown }
602         { \__template_implement_choice_elt_aux:nnn {#1} {#2} {#3} }
603         { \__template_implement_choice_elt_aux:n {#2} }
604     }
605     {
606         \clist_if_in:NnTF \l__template_tmp_clist {#2}
607         {
608             \clist_remove_all:Nn \l__template_tmp_clist {#2}
609             \__template_implement_choice_elt_aux:nnn {#1} {#2} {#3}
610         }
611         { \__template_implement_choice_elt_aux:n {#2} }
612     }
613 }

```

```

614 \cs_new_protected:Npn \__template_implement_choice_elt_aux:n #1
615 {
616   \prop_get:NVN \l__template_keytypes_prop \l__template_key_name_tl
617   \l__template_tmp_tl
618   \__template_split_keytype_arg:V \l__template_tmp_tl
619   \msg_error:nnVn { template } { unknown-choice } \l__template_key_name_tl {#1}
620 }
621 \cs_new_protected:Npn \__template_implement_choice_elt_aux:nnn #1#2#3
622 {
623   \keys_define:ne { template / #1 }
624   { \l__template_key_name_tl / #2 .code:n = { \exp_not:n {#3} } }
625   \tl_set:Nx \l__template_tmp_tl
626   { \l__template_key_name_tl \c_space_tl #2 }
627   \prop_put:NVN \l__template_vars_prop \l__template_tmp_tl {#3}
628 }
629 \cs_new_protected:Npn \__template_implement_choice_elt:n #1
630 {
631   \msg_error:nnVn { template } { choice-requires-code }
632   \l__template_key_name_tl {#1}
633 }

```

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

12.8 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.

```

634 \cs_new_protected:Npn \__template_edit_defaults:nnn #1#2#3
635 {
636   \__template_if_keys_exist:nnT {#1} {#2}
637   {
638     \__template_recover_defaults:nn {#1} {#2}
639     \__template_parse_values:nnn {#1} {#2} {#3}
640     \__template_store_defaults:nn {#1} {#2}
641   }
642 }

```

(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.

```

643 \cs_new_protected:Npn \__template_parse_values:nnn #1#2#3
644 {
645   \__template_recover_keytypes:nn {#1} {#2}
646   \keyval_parse:NNn
647   \__template_parse_values_elt:n \__template_parse_values_elt:nn {#3}
648 }

```

(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.

```

649 \cs_new_protected:Npn \__template_parse_values_elt:n #1
650 {
651   \bool_set_true:N \l__template_error_bool

```

```

652     \msg_error:nnn { template } { key-no-value } {#1}
653 }

```

(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
\_template_parse_values_elt_aux:w key name, stored in \l__template_key_name_tl.
\_template_parse_values_elt_aux:n
\_template_parse_values_exp:n
\_template_parse_values_exp:o
\_template_parse_values_exp:V
\_template_parse_values_exp:v
\_template_parse_values_exp:e
\_template_parse_values_exp:N
\_template_parse_values_exp:c
654 \cs_new_protected:Npn \_template_parse_values_elt:nn #1#2
655 {
656     \use:e
657     {
658         \_template_parse_values_elt_aux:w
659         \tl_trim_spaces:e { \tl_to_str:n { #1 : n : } }
660         \exp_not:N \q_stop
661     }
662     \prop_get:NVNTF \l__template_keytypes_prop \l__template_key_name_tl
663     \l__template_tmp_tl
664     { \_template_parse_values_elt_aux:n {#2} }
665     { \msg_error:nnV { template } { unknown-key } \l__template_key_name_tl }
666 }
667 \use:e
668 {
669     \cs_new_protected:Npn \exp_not:N \_template_parse_values_elt_aux:w
670     #1 \token_to_str:N : #2 \token_to_str:N : #3 \exp_not:N \q_stop
671 }
672 {
673     \tl_set:Nn \l__template_key_name_tl {#1}
674     \str_set:Nn \l__template_value_exp_str {#2}
675 }
676 \cs_new_protected:Npn \_template_parse_values_elt_aux:n #1
677 {
678     \_template_split_keytype_arg:V \l__template_tmp_tl
679     \cs_if_exist_use:cF { \_template_parse_values_exp: \l__template_value_exp_str }
680     {
681         \msg_error:nnV { template } { unknown-expansion } \l__template_value_exp_str
682         \use_none:n
683     }
684     {#1}
685 }
686 \cs_new_protected:Npn \_template_parse_values_exp:n #1
687 { \prop_put:Non \l__template_values_prop \l__template_key_name_tl {#1} }
688 \cs_generate_variant:Nn \_template_parse_values_exp:n { o , V , v , e }
689 \cs_new_eq:NN \_template_parse_values_exp:N \_template_parse_values_exp:n
690 \cs_generate_variant:Nn \_template_parse_values_exp:N { c }

```

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

_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.

```

691 \cs_new_protected:Npn \_template_template_set_eq:nnn #1#2#3
692 {
693     \_template_recover_defaults:nn {#1} {#3}
694     \_template_store_defaults:nn {#1} {#2}
695     \_template_recover_keytypes:nn {#1} {#3}

```

```

696     \__template_store_keytypes:nn {#1} {#2}
697     \__template_recover_vars:nn {#1} {#3}
698     \__template_store_vars:nn {#1} {#2}
699     \cs_if_exist:cT { \c__template_code_root_tl #1 / #2 }
700       { \msg_info:nnnn { template } { declare-template-code } {#1} {#2} }
701     \cs_gset_eq:cc { \c__template_code_root_tl #1 / #2 }
702       { \c__template_code_root_tl #1 / #3 }
703   }

```

(End of definition for __template_template_set_eq:nnn.)

12.9 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.

```

704 \cs_new_protected:Npn \__template_declare_instance:nnnn #1#2#3#4
705   {
706     \__template_execute_if_code_exist:nnT {#1} {#2}
707     {
708       \__template_recover_defaults:nn {#1} {#2}
709       \__template_recover_vars:nn {#1} {#2}
710       \__template_declare_instance_aux:nnnn {#1} {#2} {#3} {#4}
711     }
712   }
713 \cs_new_protected:Npn \__template_declare_instance_aux:nnnn #1#2#3#4
714   {
715     \bool_set_false:N \l__template_error_bool
716     \__template_parse_values:nnn {#1} {#2} {#4}
717     \bool_if:NF \l__template_error_bool
718     {
719       \prop_put:Nnn \l__template_values_prop { from-template } {#2}
720       \__template_store_values:nn {#1} {#3}
721       \__template_convert_to_assignments:
722       \cs_if_exist:cT { \c__template_instances_root_tl #1 / #3 }
723         { \msg_info:nnnn { template } { declare-instance } {#3} {#1} }
724       \cs_set_protected:cpe { \c__template_instances_root_tl #1 / #3 }
725         {
726           \exp_not:N \__template_assignments_push:n
727             { \exp_not:V \l__template_assignments_tl }
728           \exp_not:c { \c__template_code_root_tl #1 / #2 }
729         }
730     }
731   }

```

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

```

\__template_instance_set_eq:nnn

```

Copy-paste an instance.

```

732 \cs_new_protected:Npn \__template_instance_set_eq:nnn #1#2#3
733   {
734     \__template_if_instance_exist:nnTF {#1} {#3}

```

```

735     {
736         \__template_recover_values:nn {#1} {#3}
737         \__template_store_values:nn {#1} {#2}
738         \cs_if_exist:cT { \c__template_instances_root_tl #1 / #2 }
739         { \msg_info:nnnn { template } { declare-instance } {#2} {#1} }
740         \cs_set_eq:cc { \c__template_instances_root_tl #1 / #2 }
741         { \c__template_instances_root_tl #1 / #3 }
742     }
743     { \msg_error:nnnn { template } { unknown-instance } {#1} {#3} }
744 }

```

(End of definition for __template_instance_set_eq:nnn.)

```

\__template_edit_instance:nnn
\__template_edit_instance_aux:nnnn
\__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.

```

745 \cs_new_protected:Npn \__template_edit_instance:nnn #1#2#3
746 {
747     \__template_if_instance_exist:nnTF {#1} {#2}
748     {
749         \__template_recover_values:nn {#1} {#2}
750         \prop_get:NnN \l__template_values_prop { from-template }
751         \l__template_tmp_tl
752         \__template_edit_instance_aux:nVnn
753         {#1} \l__template_tmp_tl {#2} {#3}
754     }
755     { \msg_error:nnnn { template } { unknown-instance } {#1} {#2} }
756 }
757 \cs_new_protected:Npn \__template_edit_instance_aux:nnnn #1#2#3#4
758 {
759     \__template_recover_vars:nn {#1} {#2}
760     \__template_declare_instance_aux:nnnn {#1} {#2} {#3} {#4}
761 }
762 \cs_generate_variant:Nn \__template_edit_instance_aux:nnnn { nV }

```

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

```

\__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”.

```

763 \cs_new_protected:Npn \__template_convert_to_assignments:
764 {
765     \tl_clear:N \l__template_assignments_tl
766     \seq_map_function:NN \l__template_key_order_seq
767     \__template_convert_to_assignments_aux:n
768 }
769 \cs_new_protected:Npn \__template_convert_to_assignments_aux:n #1
770 {
771     \prop_get:NnN \l__template_keytypes_prop {#1} \l__template_tmp_tl
772     \__template_convert_to_assignments_aux:nV {#1} \l__template_tmp_tl
773 }

```

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.

```

774 \cs_new_protected:Npn \__template_convert_to_assignments_aux:nn #1#2
775 {
776   \prop_get:NnNT \l__template_values_prop {#1} \l__template_value_tl
777   {
778     \prop_get:NnNTF \l__template_vars_prop {#1} \l__template_var_tl
779     {
780       \__template_split_keytype_arg:n {#2}
781       \str_if_eq:VnF \l__template_keytype_tl { choice }
782       {
783         \str_if_eq:VnF \l__template_keytype_tl { code }
784         { \__template_find_global: }
785       }
786       \tl_set:Nn \l__template_key_name_tl {#1}
787       \cs_if_exist_use:cF { __template_assign_ \l__template_keytype_tl : }
788       { \__template_assign_variable: }
789     }
790     { \msg_error:nnn { template } { unknown-attribute } {#1} }
791   }
792 }
793 \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.

```

794 \cs_new_protected:Npn \__template_find_global:
795 {
796   \bool_set_false:N \l__template_global_bool
797   \tl_if_in:onT \l__template_var_tl { global }
798   {
799     \exp_after:wN \__template_find_global_aux:w \l__template_var_tl \s__template_stop
800   }
801 }
802 \cs_new_protected:Npn \__template_find_global_aux:w #1 global #2 \s__template_stop
803 {
804   \tl_set:Nn \l__template_var_tl {#2}
805   \bool_set_true:N \l__template_global_bool
806 }

```

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

`\__template_instance_value:nnn` For editing templates.

```

807 \cs_new:Npn \__template_instance_value:nnn #1#2#3
808 {
809   \__template_if_instance_exist:nnT {#1} {#2}
810   { \prop_item:cn { \c__template_values_root_tl #1 / #2 } {#3} }
811 }

```

(End of definition for `\__template_instance_value:nnn`.)

12.10 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.

```
812 \cs_new_protected:Npn \__template_use_template:nnn #1#2#3
813 {
814   \__template_execute_if_code_exist:nnT {#1} {#2}
815   {
816     \__template_recover_defaults:nn {#1} {#2}
817     \__template_recover_vars:nn {#1} {#2}
818     \__template_parse_values:nnn {#1} {#2} {#3}
819     \__template_convert_to_assignments:
820     \__template_debug_typeout:n
821     { Using~template~'#2'~of~type~'#1'~directly~ \msg_line_context: }
822     \use:c { \c__template_code_root_tl #1 / #2 }
823   }
824 }
```

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

12.11 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.

`__template_assign_boolean_aux:n`

```
825 \cs_new_protected:Npn \__template_assign_boolean:
826 {
827   \bool_if:NTF \l__template_global_bool
828   { \__template_assign_boolean_aux:n { bool_gset } }
829   { \__template_assign_boolean_aux:n { bool_set } }
830 }
831 \cs_new_protected:Npn \__template_assign_boolean_aux:n #1
832 {
833   \__template_if_key_value:VTF \l__template_value_tl
834   {
835     \__template_key_to_value:
836     \tl_put_right:Ne \l__template_assignments_tl
837     {
838       \exp_not:c { #1 _eq:NN }
839       \exp_not:V \l__template_var_tl
840       \exp_not:V \l__template_value_tl
841     }
842   }
843   {
844     \tl_put_right:Ne \l__template_assignments_tl
845     {
846       \exp_not:c { #1 _ \l__template_value_tl :N }
847       \exp_not:V \l__template_var_tl
848     }
849   }
850 }
```

(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
851 \cs_new_protected:Npn __template_assign_choice:
852 {
853   __template_assign_choice_aux:eF
854   { \l__template_key_name_tl \c_space_tl \l__template_value_tl }
855   {
856     __template_assign_choice_aux:eF
857     { \l__template_key_name_tl \c_space_tl unknown }
858     {
859       \prop_get:NVN \l__template_keytypes_prop \l__template_key_name_tl
860       \l__template_tmp_tl
861       __template_split_keytype_arg:V \l__template_tmp_tl
862       \msg_error:nnVV { template } { unknown-choice }
863       \l__template_key_name_tl
864       \l__template_value_tl
865     }
866   }
867 }
868 \cs_new_protected:Npn __template_assign_choice_aux:nF #1
869 {
870   \prop_get:NnNTF \l__template_vars_prop {#1} \l__template_tmp_tl
871   { \tl_put_right:NV \l__template_assignments_tl \l__template_tmp_tl }
872 }
873 \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
874 \cs_new_protected:Npn __template_assign_function:
875 {
876   \bool_if:NTF \l__template_global_bool
877   { __template_assign_function_aux:N \cs_gset_protected:Npn }
878   { __template_assign_function_aux:N \cs_set_protected:Npn }
879 }
880 \cs_new_protected:Npn __template_assign_function_aux:N #1
881 {
882   \tl_put_right:Ne \l__template_assignments_tl
883   {
884     \cs_generate_from_arg_count:NNnn
885     \exp_not:V \l__template_var_tl
886     \exp_not:N #1
887     { \exp_not:V \l__template_keytype_arg_tl }
888     { \exp_not:V \l__template_value_tl }
889   }
890 }

```

(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.

```

891 \cs_new_protected:Npn __template_assign_instance:

```

```

892 {
893   \bool_if:NTF \l__template_global_bool
894   { \__template_assign_instance_aux:N \cs_gset_protected:Npn }
895   { \__template_assign_instance_aux:N \cs_set_protected:Npn }
896 }
897 \cs_new_protected:Npn \__template_assign_instance_aux:N #1
898 {
899   \tl_put_right:Ne \l__template_assignments_tl
900   {
901     \exp_not:N #1 \exp_not:V \l__template_var_tl
902     {
903       \__template_use_instance:nn
904       { \exp_not:V \l__template_keytype_arg_tl }
905       { \exp_not:V \l__template_value_tl }
906     }
907   }
908 }

```

(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.

```

909 \cs_new_protected:Npn \__template_assign_variable:
910 {
911   \exp_args:Ne \__template_assign_variable:n
912   {
913     \__template_map_var_type:
914     -
915     \bool_if:NT \l__template_global_bool { g }
916     set:N
917   }
918 }

```

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

```

919 \cs_new_protected:Npn \__template_assign_variable:n #1
920 {
921   \__template_if_key_value:VTF \l__template_value_tl
922   {
923     \__template_key_to_value:
924     \tl_put_right:Ne \l__template_assignments_tl
925     {
926       \exp_not:c { #1 V } \exp_not:V \l__template_var_tl
927       \exp_not:V \l__template_value_tl
928     }
929   }
930   {
931     \tl_put_right:Ne \l__template_assignments_tl
932     {
933       \exp_not:c { #1 n } \exp_not:V \l__template_var_tl
934       { \exp_not:V \l__template_value_tl }
935     }
936   }
937 }

```

```

936     }
937 }

```

(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`.

```

938 \cs_new_protected:Npn __template_key_to_value:
939 { \exp_after:wN __template_key_to_value_auxi:w \l__template_value_tl }
940 \cs_new_protected:Npn __template_key_to_value_auxi:w \KeyValue #1
941 {
942   \tl_set:Nx \l__template_tmp_tl { \tl_trim_spaces:e { \tl_to_str:n {#1} } }
943   \prop_get:NVNTF \l__template_vars_prop \l__template_tmp_tl
944     \l__template_value_tl
945   {
946     \exp_after:wN __template_key_to_value_auxii:w \l__template_value_tl
947     \s__template_mark_global \q__template_nil \s__template_stop
948   }
949   { \msg_error:nnV { template } { unknown-attribute } \l__template_tmp_tl }
950 }
951 \cs_new_protected:Npn __template_key_to_value_auxii:w #1 global #2#3 \s__template_stop
952 {
953   __template_quark_if_nil:NF #2
954   { \tl_set:Nn \l__template_value_tl {#2} }
955 }

```

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

12.12 Using instances

`__template_use_instance: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.

```

956 \cs_new_protected:Npn __template_use_instance:nn #1#2
957 {
958   __template_if_use_template:nTF {#2}
959   { __template_use_instance_aux:nNnnn {#1} #2 }
960   { __template_use_instance_auxi:nn {#1} {#2} }
961 }
962 \cs_new_protected:Npn __template_use_instance_aux:nNnnn #1#2#3#4#5
963 {
964   \str_if_eq:nnTF {#1} {#3}
965   { __template_use_template:nnn {#3} {#4} {#5} }
966   { \msg_error:nnnn { template } { type-mismatch } {#1} {#3} }
967 }
968 \cs_new_protected:Npn __template_use_instance_auxi:nn #1#2
969 {
970   __template_if_instance_exist:nnTF {#1} {#2}
971   { __template_use_instance_auxii:nn {#1} {#2} }

```

```

972     { \msg_error:nnnn { template } { unknown-instance } {#1} {#2} }
973   }
974   \cs_new_protected:Npn \__template_use_instance_auxii:nn #1#2
975   {
976     \__template_debug_typeout:n
977     { Using~instance~'~#2'~of~type~'~#1'~ \msg_line_context: }
978     \use:c { \c__template_instances_root_tl #1 / #2 }
979   }

```

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

12.13 Assignment manipulation

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

__template_assignments_pop: To actually use the assignments.

```

980   \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.

```

981   \cs_new_protected:Npn \__template_assignments_push:n #1
982   { \tl_set:Nn \l__template_assignments_tl {#1} }

```

(End of definition for __template_assignments_push:n.)

12.14 Showing templates and instances

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

```

983   \cs_new_protected:Npn \__template_show_code:nn #1#2
984   { \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.

```

\__template_show_keytypes:nn
\__template_show_vars:nn
\__template_show:Nnnn
985   \cs_new_protected:Npn \__template_show_defaults:nn #1#2
986   {
987     \__template_if_keys_exist:nnT {#1} {#2}
988     {
989       \__template_recover_defaults:nn {#1} {#2}
990       \__template_show:Nnnn \l__template_values_prop
991       {#1} {#2} { default~values }
992     }
993   }
994   \cs_new_protected:Npn \__template_show_keytypes:nn #1#2
995   {
996     \__template_if_keys_exist:nnT {#1} {#2}
997     {
998       \__template_recover_keytypes:nn {#1} {#2}
999       \__template_show:Nnnn \l__template_keytypes_prop
1000       {#1} {#2} { interface }
1001     }
1002   }

```

```

1003 \cs_new_protected:Npn \__template_show_vars:nn #1#2
1004 {
1005     \__template_execute_if_code_exist:nnT {#1} {#2}
1006     {
1007         \__template_recover_vars:nn {#1} {#2}
1008         \__template_show:Nnnn \l__template_vars_prop
1009         {#1} {#2} { variable~mapping }
1010     }
1011 }
1012 \cs_new_protected:Npn \__template_show:Nnnn #1#2#3#4
1013 {
1014     \msg_show:nneeee { template } { show-attribute }
1015     { \tl_to_str:n {#2} }
1016     { \tl_to_str:n {#3} }
1017     { \tl_to_str:n {#4} }
1018     { \prop_map_function:NN #1 \msg_show_item_unbraced:nn }
1019 }

```

(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.

```

1020 \cs_new_protected:Npn \__template_show_values:nn #1#2
1021 {
1022     \__template_if_instance_exist:nnTF {#1} {#2}
1023     {
1024         \__template_recover_values:nn {#1} {#2}
1025         \msg_show:nneeee { template } { show-values }
1026         { \tl_to_str:n {#1} }
1027         { \tl_to_str:n {#2} }
1028         {
1029             \prop_map_function:NN \l__template_values_prop
1030             \msg_show_item_unbraced:nn
1031         }
1032     }
1033     { \msg_info:nnnn { template } { unknown-instance } {#1} {#2} }
1034 }

```

(End of definition for __template_show_values:nn.)

12.15 Messages

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

```

1035 \msg_new:nnnn { template } { argument-number-mismatch }
1036 { Template~type~'#1'~takes~#2~argument(s). }
1037 {
1038     Templates-of~type~'#1'~require~#2~argument(s).\
1039     You-have~tried~to~make~a~template~for~'#1'~
1040     with~#3~argument(s),~which~is~not~possible:~
1041     the~number~of~arguments~must~agree.
1042 }
1043 \msg_new:nnnn { template } { bad-number-of-arguments }
1044 { Bad-number~of~arguments~for~template~type~'#1'. }
1045 {
1046     A~template~may~accept~between~0~and~9~arguments.\

```

```

1047     You-asked-to-use~#2-arguments:~this-is-not-supported.
1048 }
1049 \msg_new:nnnn { template } { bad-variable }
1050 { Incorrect~variable~description~'#1'. }
1051 {
1052     The~argument~'#1'~is~not~of~the~form \\
1053     ~~~<variable>'\\
1054     ~or~\\
1055     ~~~global~<variable>'\\.
1056     It~must~be~given~in~one~of~these~formats~to~be~used~in~a~template.
1057 }
1058 \msg_new:nnnn { template } { choice-not-implemented }
1059 { The~choice~'#1'~has~no~implementation. }
1060 {
1061     Each~choice~listed~in~the~interface~for~a~template~must~
1062     have~an~implementation.
1063 }
1064 \msg_new:nnnn { template } { choice-no-code }
1065 { The~choice~'#1'~requires~implementation~details. }
1066 {
1067     When~creating~template~code~using~\DeclareTemplateCode,~
1068     each~choice~name~must~have~an~associated~implementation.\\
1069     This~should~be~given~after~a~'= '~sign:~LaTeX~did~not~find~one.
1070 }
1071 \msg_new:nnnn { template } { choice-requires-code }
1072 { The~choice~'#2'~for~key~'#1'~requires~an~implementation. }
1073 {
1074     You~should~have~put:\\
1075     \ \ #1::~choice~{~#2 = <code> ~} \\
1076     but~LaTeX~did~not~find~any~<code>.
1077 }
1078 \msg_new:nnnn { template } { duplicate-key-interface }
1079 { Key~'#1'~appears~twice~in~interface~definition~\msg_line_context:. }
1080 {
1081     Each~key~can~only~have~one~interface~declared~in~a~template.\\
1082     LaTeX~found~two~interfaces~for~'#1'.
1083 }
1084 \msg_new:nnnn { template } { keytype-requires-argument }
1085 { The~key~type~'#1'~requires~an~argument~\msg_line_context:. }
1086 {
1087     You~should~have~put:\\
1088     \ \ <key-name>::~#1~{~<argument>~} \\
1089     but~LaTeX~did~not~find~an~<argument>.
1090 }
1091 \msg_new:nnnn { template } { invalid-keytype }
1092 { The~key~'#1'~is~missing~a~key~type~\msg_line_context:. }
1093 {
1094     Each~key~in~a~template~requires~a~key~type,~given~in~the~form:\\
1095     \ \ <key>::~~<key-type>\\
1096     LaTeX~could~not~find~a~<key-type>~in~your~input.
1097 }
1098 \msg_new:nnnn { template } { key-no-value }
1099 { The~key~'#1'~has~no~value~\msg_line_context:. }
1100 {

```

```

1101     When~creating~an~instance~of~a~template~
1102     every~key~listed~must~include~a~value:\\
1103     \ \ <key>~==<value>
1104 }
1105 \msg_new:nnnn { template } { key-no-variable }
1106 { The~key~'#1'~requires~implementation~details~\msg_line_context:. }
1107 {
1108     When~creating~template~code~using~\DeclareTemplateCode,~
1109     each~key~name~must~have~an~associated~implementation.\\
1110     This~should~be~given~after~a~'='~sign:~LaTeX~did~not~find~one.
1111 }
1112 \msg_new:nnnn { template } { key-not-implemented }
1113 { Key~'#1'~has~no~implementation~\msg_line_context:. }
1114 {
1115     The~definition~of~key~implementations~for~template~'#2'~
1116     of~template~type~'#3'~does~not~include~any~details~for~key~'#1'.\\
1117     The~key~was~declared~in~the~interface~definition,~
1118     and~so~an~implementation~is~required.
1119 }
1120 \msg_new:nnnn { template } { missing-keytype }
1121 { The~key~'#1'~is~missing~a~key~type~\msg_line_context:. }
1122 {
1123     Key~interface~definitions~should~be~of~the~form\\
1124     \ \ #1::~<key-type>\\
1125     but~LaTeX~could~not~find~a~<key-type>.
1126 }
1127 \msg_new:nnnn { template } { no-template-code }
1128 {
1129     The~template~'#2'~of~type~'#1'~is~unknown~
1130     or~has~no~implementation.
1131 }
1132 {
1133     There~is~no~code~available~for~the~template~name~given.\\
1134     This~should~be~given~using~\DeclareTemplateCode.
1135 }
1136 \msg_new:nnnn { template } { type-already-defined }
1137 { Template~type~'#1'~already~defined. }
1138 {
1139     You~have~used~\NewTemplateType~
1140     with~a~template~type~that~has~already~been~defined.
1141 }
1142 \msg_new:nnnn { template } { type-mismatch }
1143 { Template~types~'#1'~and~'#2'~do~not~agree. }
1144 {
1145     You~are~trying~to~use~a~template~directly~with~\UseInstance
1146     (or~a~similar~function),~but~the~template~types~do~not~match.
1147 }
1148 \msg_new:nnnn { template } { unknown-attribute }
1149 { The~template~attribute~'#1'~is~unknown. }
1150 {
1151     There~is~a~definition~in~the~current~template~reading\\
1152     \ \ \token_to_str:N \KeyValue {~#1~} \\
1153     but~there~is~no~key~called~'#1'.
1154 }

```

```

1155 \msg_new:nnnn { template } { unknown-choice }
1156 { The-choice~'#2'~was~not~declared~for~key~'#1'. }
1157 {
1158     The-key~'#1'~takes~a~fixed~list~of~choices~
1159     and~this~list~does~not~include~'#2'.
1160 }
1161 \msg_new:nnnn { template } { unknown-default-choice }
1162 { The-default-choice~'#2'~was~not~declared~for~key~'#1'. }
1163 {
1164     The-key~'#1'~takes~a~fixed~list~of~choices~
1165     and~this~list~does~not~include~'#2'.
1166 }
1167 \msg_new:nnnn { template } { unknown-expansion }
1168 { The-expansion~type~'#1'~is~unknown. }
1169 {
1170     Key-values~can~only~be~expanded~using~one~of~the~pre-defined~methods:~
1171     n,~o,~V,~v,~e,~N~or~c.
1172 }
1173 \msg_new:nnnn { template } { unknown-instance }
1174 { The-instance~'#2'~of~type~'#1'~is~unknown. }
1175 {
1176     You~have~asked~to~use~an~instance~'#2',~
1177     but~this~has~not~been~created.
1178 }
1179 \msg_new:nnnn { template } { unknown-key }
1180 { Unknown~template~key~'#1'. }
1181 {
1182     The-key~'#1'~was~not~declared~in~the~interface~
1183     for~the~current~template.
1184 }
1185 \msg_new:nnnn { template } { unknown-keytype }
1186 { The-key-type~'#1'~is~unknown. }
1187 {
1188     Valid-key-types~are:\\
1189     --boolean;\\
1190     --choice;\\
1191     --commalist;\\
1192     --function;\\
1193     --instance;\\
1194     --integer;\\
1195     --length;\\
1196     --muskip;\\
1197     --real;\\
1198     --skip;\\
1199     --tokenlist.
1200 }
1201 \msg_new:nnnn { template } { unknown-type }
1202 { The-template-type~'#1'~is~unknown. }
1203 {
1204     A~template~type~needs~to~be~defined~with~\NewTemplateType
1205     prior~to~using~it.
1206 }
1207 \msg_new:nnnn { template } { unknown-template }
1208 { The-template~'#2'~of~type~'#1'~is~unknown. }

```

```

1209 {
1210   No~interface~has~been~declared~for~a~template~
1211   '#2'~of~template~type~'#1'.
1212 }

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

1213 \msg_new:nnn { template } { declare-instance }
1214 { Declaring~instance~~'#1'~of~type~'#2'~\msg_line_context:. }
1215 \msg_new:nnn { template } { declare-template-code }
1216 { Declaring~code~for~template~'#2'~of~template~type~'#1'~\msg_line_context:. }
1217 \msg_new:nnn { template } { declare-template-interface }
1218 {
1219   Declaring~interface~for~template~'#2'~of~template~type~'#1'~
1220   \msg_line_context:.
1221 }
1222 \msg_new:nnn { template } { declare-type }
1223 { Declaring~template~type~'#1'~taking~#2~argument(s)~\msg_line_context:. }
1224 \msg_new:nnn { template } { show-attribute }
1225 {
1226   The~template~'#2'~of~type~'#1'~has~
1227   \tl_if_empty:nTF {#4} { no~#3. } { #3 : #4 }
1228 }
1229 \msg_new:nnn { template } { show-values }
1230 {
1231   The~instance~'#2'~of~type~'#1'~has~
1232   \tl_if_empty:nTF {#3} { no~values. } { values: #3 }
1233 }

Also add template to the LaTeX messages.
1234 \prop_gput:Nnn \g_msg_module_type_prop { template } { LaTeX }

```

12.16 User functions

All simple translations.

```

\NewTemplateType
\DeclareTemplateInterface
\DeclareTemplateCode
\DeclareTemplateCopy
\EditTemplateDefaults
\UseTemplate
\DeclareInstance
\DeclareInstanceCopy
\EditInstance
\UseInstance

1235 \cs_new_protected:Npn \NewTemplateType #1#2
1236 { \__template_define_type:nn {#1} {#2} }
1237 \cs_new_protected:Npn \DeclareTemplateInterface #1#2#3#4
1238 { \__template_declare_template_keys:nnnn {#1} {#2} {#3} {#4} }
1239 \cs_new_protected:Npn \DeclareTemplateCode #1#2#3#4#5
1240 { \__template_declare_template_code:nnnnn {#1} {#2} {#3} {#4} {#5} }
1241 \cs_new_protected:Npn \DeclareTemplateCopy #1#2#3
1242 { \__template_template_set_eq:nnn {#1} {#2} {#3} }
1243 \cs_new_protected:Npn \EditTemplateDefaults #1#2#3
1244 { \__template_edit_defaults:nnn {#1} {#2} {#3} }
1245 \cs_new_protected:Npn \UseTemplate #1#2#3
1246 { \__template_use_template:nnn {#1} {#2} {#3} }
1247 \cs_new_protected:Npn \DeclareInstance #1#2#3#4
1248 { \__template_declare_instance:nnnn {#1} {#3} {#2} {#4} }
1249 \cs_new_protected:Npn \DeclareInstanceCopy #1#2#3
1250 { \__template_instance_set_eq:nnn {#1} {#2} {#3} }
1251 \cs_new_protected:Npn \EditInstance #1#2#3
1252 { \__template_edit_instance:nnn {#1} {#2} {#3} }
1253 \cs_new_protected:Npn \UseInstance #1#2
1254 { \__template_use_instance:nn {#1} {#2} }

```

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

`\ShowTemplateCode` The show functions are again just translation.
`\ShowTemplateDefaults` 1255 `\cs_new_protected:Npn \ShowTemplateCode #1#2`
`\ShowTemplateInterface` 1256 `{ \__template_show_code:nn {#1} {#2} }`
`\ShowTemplateVariables` 1257 `\cs_new_protected:Npn \ShowTemplateDefaults #1#2`
`\ShowInstanceValues` 1258 `{ \__template_show_defaults:nn {#1} {#2} }`
1259 `\cs_new_protected:Npn \ShowTemplateInterface #1#2`
1260 `{ \__template_show_keytypes:nn {#1} {#2} }`
1261 `\cs_new_protected:Npn \ShowTemplateVariables #1#2`
1262 `{ \__template_show_vars:nn {#1} {#2} }`
1263 `\cs_new_protected:Npn \ShowInstanceValues #1#2`
1264 `{ \__template_show_values:nn {#1} {#2} }`

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

`\IfInstanceExistsT` More direct translation.
`\IfInstanceExistsF` 1265 `\cs_new:Npn \IfInstanceExistsTF #1#2`
`\IfInstanceExistsTF` 1266 `{ \__template_if_instance_exist:nnTF {#1} {#2} }`
1267 `\cs_new:Npn \IfInstanceExistsT #1#2`
1268 `{ \__template_if_instance_exist:nnT {#1} {#2} }`
1269 `\cs_new:Npn \IfInstanceExistsF #1#2`
1270 `{ \__template_if_instance_exist:nnF {#1} {#2} }`

(End of definition for `\IfInstanceExistsT`, `\IfInstanceExistsF`, and `\IfInstanceExistsTF`. These functions are documented on page 9.)

`\KeyValue` Simply dump the argument when executed: this should not happen.
1271 `\cs_new_protected:Npn \KeyValue #1 {#1}`

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

`\InstanceValue` 1272 `\cs_new:Npn \InstanceValue #1#2#3`
1273 `{ \__template_instance_value:nnn {#1} {#2} {#3} }`

(End of definition for `\InstanceValue`. This function is documented on page 9.)

`\AssignTemplateKeys` A short call to use a token register by proxy.
1274 `\cs_new_protected:Npn \AssignTemplateKeys { \__template_assignments_pop: }`

(End of definition for `\AssignTemplateKeys`. This function is documented on page 6.)

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

1275 `\cs_new_protected:Npn \SetKnownTemplateKeys #1#2#3`
1276 `{`
1277 `\tl_if_empty:OTF {#3}`
1278 `{`
1279 `\tl_set_eq:NN \UnusedTemplateKeys \c_empty_tl`
1280 `}`
1281 `{`
1282 `\keys_set_known:noN { template / #1 / #2 } {#3} \UnusedTemplateKeys`
1283 `}`
1284 `}`

```

1285 \cs_new_protected:Npn \SetTemplateKeys #1#2#3
1286 {
1287   \tl_if_empty:of {#3}
1288   {
1289     \keys_set:no { template / #1 / #2 } {#3}
1290   }
1291 }
1292 \tl_new:N \UnusedTemplateKeys

```

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

`\DebugTemplatesOn`
`\DebugTemplatesOff`

```

1293 \cs_new_protected:Npn \DebugTemplatesOn { \__template_debug_on: }
1294 \cs_new_protected:Npn \DebugTemplatesOff { \__template_debug_off: }

```

(End of definition for `\DebugTemplatesOn` and `\DebugTemplatesOff`. These functions are documented on page 11.)

```

1295 <latexrelease>\IncludeInRelease{0000/00/00}{ltemplates}%
1296 <latexrelease>           {Prototype~document~commands}%
1297 <latexrelease>
1298 <latexrelease>\EndModuleRelease
1299 \ExplSyntaxOff
1300 </2ekernel | latexrelease>

```

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

```

1301 <@@=

```

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	
<code>\</code>	1038, 1046, 1052, 1053, 1054, 1055, 1068, 1074, 1075, 1081, 1087, 1088, 1094, 1095, 1102, 1109, 1116, 1123, 1124, 1133, 1151, 1152, 1188, 1189, 1190, 1191, 1192, 1193, 1194, 1195, 1196, 1197, 1198
<code>_</code>	1075, 1088, 1095, 1103, 1124, 1152
A	
<code>\AssignTemplateKeys</code>	1274, 398, 403, 6
B	
bool commands:	
<code>\bool_if:N</code>	876, 893, 915, 253, 259, 717, 827
<code>\bool_new:N</code>	38, 39
<code>\bool_set_false:N</code>	308, 715, 796
<code>\bool_set_true:N</code>	279, 321, 651, 805
<code>\l_tmpa_bool</code>	5
C	
<code>\c</code>	394
<code>\caption</code>	2
clist commands:	
<code>\clist_if_empty:N</code>	570, 599
<code>\clist_if_in:Nn</code>	297, 606
<code>\clist_map_inline:Nn</code>	572
<code>\clist_new:N</code>	50
<code>\clist_remove_all:Nn</code>	608
<code>\clist_set:Nn</code>	560
<code>\l_tmpa_clist</code>	5
cs commands:	
<code>\cs_generate_from_arg_count:NNnn</code>	884, 413, 476

\msg_error:nnnn	862, 966, 972, 87, 106, 230, 591, 619, 631, 743, 755
\msg_error:nnnnn	79, 426
\msg_info:nnnn	1033, 140, 224, 412, 700, 723, 739
\msg_line_context:	977, 1079, 1085, 1092, 1099, 1106, 1113, 1121, 1214, 1216, 1220, 1223, 821
\g_msg_module_type_prop	1234
\msg_new:nnn	1213, 1215, 1217, 1222, 1224, 1229
\msg_new:nnnn	1035, 1043, 1049, 1058, 1064, 1071, 1078, 1084, 1091, 1098, 1105, 1112, 1120, 1127, 1136, 1142, 1148, 1155, 1161, 1167, 1173, 1179, 1185, 1201, 1207
\msg_show:nnnnn	1025
\msg_show:nnnnnn	1014
\msg_show_item_unbraced:nn	1018, 1030
muskip commands:	
\muskip_new:N	53
\l_tmpa_muskip	5
N	
\NewModuleRelease	7
\NewTemplateType	1139, 1204, 1235, 3
P	
prg commands:	
\prg_generate_conditional_variant:Nnn	114
\prg_new_conditional:Npnn	108, 115, 121
\prg_return_false:	112, 119, 125
\prg_return_true:	111, 118, 124
prop commands:	
\prop_clear:N	175, 185, 201, 211, 240, 241, 421
\prop_clear_new:N	154
\prop_gclear:N	141
\prop_gclear_new:N	130, 162
\prop_get:NnN	859, 75, 586, 589, 616, 750, 771
\prop_get:NnNTF	870, 943, 434, 567, 662, 776, 778
\prop_gput:Nnn	1234, 226
\prop_gset_eq:NN	131, 144, 163
\prop_if_exist:NTF	138, 169, 179, 195, 205
\prop_if_in:NnTF	98, 215, 580, 584
\prop_item:Nn	810
\prop_map_function:NN	1018, 1029
\prop_map_inline:Nn	425
\prop_new:N	35, 46, 48, 49, 143
\prop_put:Nnn	292, 304, 484, 498, 533, 561, 627, 687, 719
\prop_remove:Nn	440
\prop_set_eq:NN	155, 172, 182, 198, 208
Q	
quark commands:	
\q_nil	110, 123
\quark_new:N	58
\q_stop	110, 123, 660, 670
quark internal commands:	
\q_template_nil	947, 58, 14
R	
regex commands:	
\regex_match:nnTF	394
S	
scan commands:	
\scan_new:N	56, 57
scan internal commands:	
\s_template_mark	947, 56, 14
\s_template_stop	947, 951, 57, 317, 328, 339, 360, 374, 382, 446, 449, 503, 508, 799, 802, 14
\section	2
seq commands:	
\seq_clear:N	191, 242
\seq_const_from_clist:Nn	16, 33
\seq_gclear_new:N	146
\seq_gset_eq:NN	147
\seq_if_exist:NTF	186
\seq_if_in:NnTF	91, 261
\seq_map_break:	280, 371
\seq_map_function:NN	257, 377, 766
\seq_new:N	47
\seq_put_right:Nn	294
\seq_set_eq:NN	188
\SetKnownTemplateKeys	1275, 6
\SetTemplateKeys	1275, 7
\ShowInstanceValues	1255, 10
\ShowTemplateCode	1255, 10
\ShowTemplateDefaults	1255, 10
\ShowTemplateInterface	1255, 10
\ShowTemplateVariables	1255, 11
skip commands:	
\skip_new:N	54
\l_tmpa_skip	5
str commands:	
\str_case:nn	546
\str_case:nnTF	465
\str_if_eq:nnTF	964, 110, 123, 274, 295, 479, 494, 530, 601, 781, 783
\str_if_in:nnTF	392

\str_new:N	45	\l__template_default_tl	37, 13
\str_set:Nn	674	\c__template_defaults_root_tl	10, 130, 131, 170, 173, 12
T			
template internal commands:			
__template_assign_boolean:	825, 825	__template_define_type:nn	1236, 213, 213
__template_assign_boolean_aux:n	825, 828, 829, 831	__template_edit_defaults:nnn	1244, 634, 634
__template_assign_choice:	851, 851	__template_edit_instance:nnn	1252, 745, 745
__template_assign_choice_-aux:nTF	851, 853, 856, 868, 873	__template_edit_instance_-aux:nnnn	752, 757, 762
__template_assign_function:	874, 874	__template_edit_instance_-aux:nnnnn	745
__template_assign_function_-aux:N	874, 877, 878, 880	\l__template_error_bool	38, 253, 259, 279, 308, 321, 651, 715, 717, 13
__template_assign_instance_-aux:N	891, 891	__template_execute_if_arg_-agree:nnTF	73, 73, 238, 387
__template_assign_variable:	891, 894, 895, 897	__template_execute_if_code_-exist:nnTF	1005, 83, 83, 706, 814
__template_assign_variable:n	909, 909, 788	__template_execute_if_keys_-exist:nnTF	102
__template_assign_variable:n	909, 911, 919	__template_execute_if_keytype_-exist:nTF	89, 89, 95, 255
__template_assignments_pop:	980, 980, 1274	__template_execute_if_type_-exist:nTF	96, 96, 236, 385
__template_assignments_push:n	981, 981, 726	__template_find_global:	784, 794, 794
\l__template_assignments_tl	871, 882, 899, 924, 931, 980, 982, 36, 727, 765, 836, 844, 13	__template_find_global_aux:w	794, 799, 802
\c__template_code_root_tl	9, 984, 85, 411, 414, 699, 701, 702, 728, 822, 12	\l__template_global_bool	876, 893, 915, 39, 796, 805, 827, 13
__template_convert_to_assignments:	721, 763, 763, 819	__template_if_instance_exist:nn	115
__template_convert_to_assignments_-aux:n	763, 767, 769	__template_if_instance_exist:nnTF	970, 1022, 1266, 1268, 1270, 115, 734, 747, 809
__template_convert_to_assignments_-aux:nn	763, 772, 774, 793	__template_if_key_value:n	108, 114
__template_debug:n	62, 68, 71, 71	__template_if_key_value:nTF	921, 108, 833
__template_debug_off:	1294, 60, 66	__template_if_keys_exist:nnTF	987, 996, 102, 389, 636
__template_debug_on:	1293, 60, 60	__template_if_use_template:n	121
__template_debug_typeof:n	976, 63, 69, 71, 72, 820	__template_if_use_template:nTF	958, 121
__template_declare_instance:nnnn	1248, 704, 704	__template_implement_choice_-elt:n	564, 597, 629
__template_declare_instance_-aux:nnnn	704, 710, 713, 760	__template_implement_choice_-elt:nnn	565, 597, 597
__template_declare_template_-code:nnnn	383, 395, 397, 402, 409	__template_implement_choice_-elt_aux:n	597, 603, 611, 614
__template_declare_template_-code:nnnnn	1240, 383, 383	__template_implement_choice_-elt_aux:nnn	597, 602, 609, 621
__template_declare_template_-keys:nnnn	1238, 234, 234	__template_implement_choices:nn	467, 558, 558
__template_declare_type:nn	213, 217, 219	__template_implement_choices_-default:	558, 569, 576

```

\__template_instance_set_eq:nnn .
    ..... 1250, 732, 732
\__template_instance_value:nnn ..
    ..... 1273, 807, 807
\c__template_instances_root_tl ..
    ..... 11,
    978, 117, 722, 724, 738, 740, 741, 12
\l__template_key_name_tl .....
    ..... 854, 857, 859, 863, 40,
    262, 265, 292, 294, 304, 315, 330,
    337, 342, 432, 435, 440, 485, 499,
    534, 542, 561, 562, 567, 579, 583,
    586, 589, 592, 593, 616, 619, 624,
    626, 632, 662, 665, 673, 687, 786, 13
\c__template_key_order_root_tl ..
    ..... 13, 146, 147, 186, 189, 12
\l__template_key_order_seq .. 47,
    148, 188, 191, 242, 261, 294, 766, 13
\__template_key_to_value: .....
    ..... 923, 938, 938, 835
\__template_key_to_value_auxi:w .
    ..... 938, 939, 940
\__template_key_to_value_auxii:w
    ..... 938, 946, 951
\l__template_keytype_arg_tl ....
    ..... 887, 904, 42, 276,
    289, 290, 297, 354, 368, 480, 560, 13
\l__template_keytype_tl .....
    . 41, 255, 274, 288, 295, 353, 364,
    436, 438, 465, 546, 781, 783, 787, 13
\c__template_keytypes_arg_seq ...
    ..... 33, 257, 377, 12
\l__template_keytypes_prop . 859,
    999, 46, 145, 182, 185, 241, 292,
    425, 434, 440, 586, 616, 662, 771, 13
\c__template_keytypes_root_tl 12,
    104, 138, 141, 143, 144, 180, 183, 12
\c__template_keytypes_seq . 16, 91, 12
\__template_map_var_type: .....
    ..... 913, 526, 529, 544, 544
\__template_parse_keys_elt:n ...
    ..... 244, 250, 250, 303, 20
\__template_parse_keys_elt:nn ...
    ..... 244, 301, 301
\__template_parse_keys_elt_aux: .
    ..... 250, 267, 284
\__template_parse_keys_elt_aux:n
    ..... 250, 258, 272
\__template_parse_values:nnn ...
    ..... 639, 643, 643, 716, 818
\__template_parse_values_elt:n ..
    ..... 647, 649, 649
\__template_parse_values_elt:nn .
    ..... 647, 654, 654
\__template_parse_values_elt_-
    aux:n ..... 654, 664, 676
\__template_parse_values_elt_-
    aux:w ..... 654, 658, 669
\__template_parse_values_exp:N ..
    ..... 654, 689, 690
\__template_parse_values_exp:n ..
    ..... 654, 686, 688, 689
\__template_parse_vars_elt:n ...
    ..... 423, 428, 428
\__template_parse_vars_elt:nnn ..
    ..... 423, 430, 430
\__template_parse_vars_elt_-
    auxi:nn ..... 430, 439, 444
\__template_parse_vars_elt_-
    auxii:nw ..... 430, 446, 448
\__template_parse_vars_elt_-
    auxiii:nnn . 430, 452, 456, 463, 506
\__template_parse_vars_elt_-
    auxiv:nnw ..... 430, 503, 507
\__template_parse_vars_elt_-
    auxv:nnn ... 430, 511, 515, 521, 538
\__template_parse_vars_elt_-
    key:nn ..... 430, 472, 489, 527, 539
\__template_quark_if_nil:N ..... 59
\__template_quark_if_nil:NTF .. 953
\__template_quark_if_nil:nTF ... 59
\__template_quark_if_nil_p:n ... 59
\__template_recover_defaults:nn .
    989, 167, 167, 419, 638, 693, 708, 816
\__template_recover_keytypes:nn .
    ..... 998, 167, 177, 420, 645, 695
\__template_recover_values:nn ...
    ..... 1024, 167, 193, 736, 749
\__template_recover_vars:nn ....
    ... 1007, 167, 203, 697, 709, 759, 817
\c__template_restrict_root_tl ... 12
\__template_show:Nnnn .....
    ..... 985, 990, 999, 1008, 1012
\__template_show_code:nn .....
    ..... 983, 983, 1256
\__template_show_defaults:nn ...
    ..... 985, 985, 1258
\__template_show_keytypes:nn ...
    ..... 985, 994, 1260
\__template_show_values:nn .....
    ..... 1020, 1020, 1264
\__template_show_vars:nn .....
    ..... 985, 1003, 1262
\__template_split_keytype:n ....
    ..... 252, 306, 306
\__template_split_keytype_arg:n .
    ..... 861, 347,
    351, 351, 380, 438, 588, 618, 678, 780

```

__template_split_keytype_arg_-	132, 156, 172, 175, 198, 201, 240,
aux:n	304, 567, 589, 687, 719, 750, 776, 13
__template_split_keytype_arg_-	\c__template_values_root_tl
aux:w	 14, 154, 155, 196, 199, 810, 12
__template_split_keytype_aux:w .	\l__template_var_tl
.	 885, 901, 926, 933,
__template_store_defaults:nn . .	44, 778, 797, 799, 804, 839, 847, 13
.	\l__template_vars_prop . . 870, 943,
__template_store_key_implementation:nnn	1008, 49, 164, 208, 211, 421, 484,
.	498, 533, 561, 580, 584, 627, 778, 13
__template_store_keytypes:nn . .	\c__template_vars_root_tl
.	 15, 162, 163, 206, 209, 12
__template_store_values:nn	tl commands:
.	\c_empty_tl 1279
__template_store_vars:nn	\c_space_tl . . . 854, 857, 579, 583, 626
.	\tl_clear:N 315, 354, 765
__template_template_set_eq:nnn .	\tl_const:Nn . . 9, 10, 11, 12, 13, 14, 15
.	\tl_head:w 110, 123
\l__template_tmp_clist	\tl_if_blank:nTF
. . 50, 560, 570, 572, 599, 606, 608, 14	 362, 366, 451, 454, 510, 513
\l__template_tmp_dim 51, 14	\tl_if_empty:NTF 276, 289, 342
\l__template_tmp_int	\tl_if_empty:nTF 1227, 1232, 1277, 1287
. 52, 221, 222, 225, 227, 231, 14	\tl_if_in:nnTF 311, 335, 357, 797
\l__template_tmp_muskip 53, 14	\tl_if_single:nTF 523
\l__template_tmp_skip 54, 14	\tl_new:N
\l__template_tmp_tl 860,	. . . 36, 37, 40, 41, 42, 1292, 43, 44, 55
861, 870, 871, 942, 943, 949, 55,	\tl_put_right:Nn 871,
75, 76, 80, 286, 293, 309, 310, 311,	882, 899, 924, 931, 330, 337, 836, 844
317, 568, 578, 579, 580, 582, 583,	\tl_replace_all:Nnn 310
584, 587, 588, 590, 617, 618, 625,	\tl_set:Nn
627, 663, 678, 751, 753, 771, 772, 14	. 942, 954, 982, 286, 309, 353, 364,
\g__template_type_prop	368, 432, 578, 582, 625, 673, 786, 804
. 35, 75, 98, 215, 226, 12	\tl_set_eq:NN 1279
__template_use_instance:nn	\tl_to_str:n . . 942, 18, 1015, 1016,
. 903, 956, 956, 1254	1017, 1026, 1027, 91, 333, 433, 659
__template_use_instance_-	\tl_trim_spaces:n
aux:nNnn 956, 959, 962	942, 332, 353, 365, 433, 457, 516, 659
__template_use_instance_auxi:nn	\l_tmpa_tl 5
. 956, 960, 968	token commands:
__template_use_instance_-	\token_to_str:N
auxii:nn 956, 971, 974	1152, 310, 311, 328, 335, 338, 345, 670
__template_use_template:nnn . . .	
. 965, 1246, 812, 812	U
\l__template_value_exp_str	\UnusedTemplateKeys . . 1279, 1282, 1292, 6
. 45, 674, 679, 681, 13	use commands:
\l__template_value_tl . . 854, 864,	\use:N 978, 526, 822
888, 905, 921, 927, 934, 939, 944,	\use:n 325, 369, 656, 667
946, 954, 43, 776, 833, 840, 846, 13	\use_i:nn 5
\l__template_values_prop	\use_none:n 682
. 990, 1029, 48,	\UseInstance 1145, 1235, 495, 9
	\UseTemplate 1235, 123, 9