

The `xtemplate` package

Prototype document functions*

The L^AT_EX3 Project[†]

Released 2012/09/05

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

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

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

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

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

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

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

The idea behind `xtemplate` 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. `xtemplate` also makes it easier for L^AT_EX programmers to provide their own customisations on top of a pre-existing class.

*This file describes v4205, last revised 2012/09/05.

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

1 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 itemised 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 object types, templates, and instances, and they are discussed below in sections 3, 4, and 6, respectively.

2 Objects, templates, and instances

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

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

3 Object types

An *object type* (sometimes just “object”) 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 object, for example, might take three inputs: “title”, “short title”, and “label”.

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

\DeclareObjectType**\DeclareObjectType** {*<object type>*} {*<no. of args>*}

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

\DeclareObjectType{sectioning}{3}

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

4 Templates

A *template* is a generalised design solution for representing the information of a specified object type. Templates that do the same thing, but in different ways, are grouped together by their object 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**.

\DeclareTemplateInterface

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

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

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

Key-type	Description of input
<code>boolean</code>	<code>true</code> or <code>false</code>
<code>choice{\langle choices \rangle}</code>	A list of pre-defined $\langle choices \rangle$
<code>code</code>	Generalised key type: use <code>#1</code> as the input to the key
<code>commalist</code>	A comma-separated list
<code>function{\langle N \rangle}</code>	A function definition with N arguments (N from 0 to 9)
<code>instance{\langle name \rangle}</code>	An instance of type $\langle name \rangle$
<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 { \langle key name \rangle }`

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 { object } { 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 5)
<code>code</code>	<code><code></code> using <code>#1</code> as input to the key
<code>commalist</code>	Comma list, <i>e.g.</i> <code>\l_tmpa_clist</code>
<code>function</code>	Function taking N arguments, <i>e.g.</i> <code>\use_i:nn</code>
<code>instance</code>	
<code>integer</code>	Integer variable, <i>e.g.</i> <code>\l_tmpa_int</code>
<code>length</code>	Dimension variable, <i>e.g.</i> <code>\l_tmpa_dim</code>
<code>muskip</code>	Muskip variable, <i>e.g.</i> <code>\l_tmpa_muskip</code>
<code>real</code>	Floating-point variable, <i>e.g.</i> <code>\l_tmpa_fp</code>
<code>skip</code>	Skip variable, <i>e.g.</i> <code>\l_tmpa_skip</code>
<code>tokenlist</code>	Token list variable, <i>e.g.</i> <code>\l_tmpa_tl</code>

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

<code>\DeclareTemplateCode</code>	<code><key1> = <variable1>,</code> <code><key2> = <variable2>,</code> <code><key3> = global <variable3>,</code> <code><key4> = global <variable4>,</code> <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 `<object 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>`.

With the exception of the choice, code and function key types, the `<variable>` here should be the name of an existing L^AT_EX3 register. As illustrated, the key word “global” may be included in the listing to indicate that the `<variable>` should be assigned globally. 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 object type.

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

In the final argument of `\DeclareTemplateCode` the assignment of keys defined by the template is carried out by using the function `\AssignTemplateKeys`. Thus no keys are assigned if this is missing from the `<code>` used.

`\EvaluateNow`

`\EvaluteNow {⟨expression⟩}`

The standard method when creating an instance from a template is to evaluate the *⟨expression⟩* when the instance is used. However, it may be desirable to calculate the value when declared, which can be forced using `\EvaluateNow`. Currently, this functionality is regarded as experimental: the team have not found an example where it is actually needed, and so it may be dropped *if* no good examples are suggested!

5 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 favoured, with the choice type reserved for keys which take arbitrary values.

6 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 centred 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 centred and set in 12 pt italic with a 10 pt skip before and a 12 pt skip after it”. Therefore, an instance is just a frozen version of a template with specific settings as chosen by the designer.

\DeclareInstance

```
\DeclareInstance
  {\langle object type \rangle} {\langle instance \rangle} {\langle template \rangle} {\langle parameters \rangle}
```

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

As a practical example, consider an object type for document sections (which might include chapters, parts, sections, *etc.*), which is called **sectioning**. One possible template for this object 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.

7 Document interface

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

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

<hr/> <hr/>	\UseTemplate $\{\langle object\ type\rangle\}\ \{\langle template\rangle\}$ $\{\langle settings\rangle\}\ \langle arguments\rangle$ Uses the $\langle template\rangle$ of the specified $\langle object\ type\rangle$, applying the $\langle settings\rangle$ and absorbing $\langle arguments\rangle$ as detailed by the $\langle object\ type\rangle$ 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. It is therefore useful where a template needs to be used once.
-------------	---

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

```

\DeclareInstance { object } { template } { instance }
{
  instance-key =
    \UseTemplate { object2 } { template2 } { <settings> }
}

```

8 Changing existing definitions

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

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

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

9 When template parameters should be frozen

A class designer may be inheriting templates declared by someone else, either third-party code or the L^AT_EX kernel itself. Sometimes these templates will be overly general for the purposes of the document. The user should be able to customise parts of the template instances, but otherwise be restricted to only those parameters allowed by the designer.

`\DeclareRestrictedTemplate`

`\DeclareRestrictedTemplate`
`{⟨object type⟩} {⟨parent template⟩} {⟨new template⟩}`
`{⟨parameters⟩}`

Creates a copy of the *⟨parent template⟩* for the *⟨object type⟩* called *⟨new template⟩*. The key–value list of *⟨parameters⟩* applies in the *⟨new template⟩* and cannot be changed when creating an instance.

10 Getting information about templates and instances

`\ShowInstanceValues`

`\ShowInstanceValues {⟨object type⟩} {⟨instance⟩}`

Shows the *⟨values⟩* for an *⟨instance⟩* of the given *⟨object type⟩* at the terminal.

`\ShowCollectionInstanceValues`

`\ShowInstanceValues {⟨collection⟩} {⟨object type⟩} {⟨instance⟩}`

Shows the *⟨values⟩* for an *⟨instance⟩* within a *⟨collection⟩* of the given *⟨object type⟩* at the terminal.

`\ShowTemplateCode`

`\ShowTemplateCode {⟨object type⟩} {⟨template⟩}`

Shows the *⟨code⟩* of a *⟨template⟩* for an *⟨object type⟩* in the terminal.

`\ShowTemplateDefaults`

`\ShowTemplateDefaults {⟨object type⟩} {⟨template⟩}`

Shows the *⟨default⟩* values of a *⟨template⟩* for an *⟨object type⟩* in the terminal.

`\ShowTemplateInterface`

`\ShowTemplateInterface {⟨object type⟩} {⟨template⟩}`

Shows the *⟨keys⟩* and associated *⟨key types⟩* of a *⟨template⟩* for an *⟨object type⟩* in the terminal.

`\ShowTemplateVariables`

`\ShowTemplateVariables {<object type>} {<template>}`

Shows the *<variables>* and associated *<keys>* of a *<template>* for an *<object 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 object 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 xtemplate Implementation

```
1 <*package>
2 <@@=xtemplate>
3 \ProvidesExplPackage
4   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
```

11.1 Variables and constants

```
\c__xtemplate_code_root_tl
\c__xtemplate_defaults_root_tl
\c__xtemplate_instances_root_tl
\c__xtemplate_keytypes_root_tl
\c__xtemplate_key_order_root_tl
\c__xtemplate_restrict_root_tl
\c__xtemplate_values_root_tl
\c__xtemplate_vars_root_tl

5 \tl_const:Nn \c__xtemplate_code_root_tl { template-code~>~ }
6 \tl_const:Nn \c__xtemplate_defaults_root_tl { template-defaults~>~ }
7 \tl_const:Nn \c__xtemplate_instances_root_tl { template-instance~>~ }
8 \tl_const:Nn \c__xtemplate_keytypes_root_tl { template-key~types~>~ }
9 \tl_const:Nn \c__xtemplate_key_order_root_tl { template-key~order~>~ }
10 \tl_const:Nn \c__xtemplate_restrict_root_tl { template-restrictions~>~ }
11 \tl_const:Nn \c__xtemplate_values_root_tl { template-values~>~ }
12 \tl_const:Nn \c__xtemplate_vars_root_tl { template-vars~>~ }
```

(End definition for `\c__xtemplate_code_root_tl` This function is documented on page ??.)

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

```
13 \seq_new:N \c__xtemplate_keytypes_arg_seq
14 \seq_put_right:Nn \c__xtemplate_keytypes_arg_seq { choice }
15 \seq_put_right:Nn \c__xtemplate_keytypes_arg_seq { function }
16 \seq_put_right:Nn \c__xtemplate_keytypes_arg_seq { instance }
(End definition for \c__xtemplate_keytypes_arg_seq This variable is documented on page ??.)
```

`\g__xtemplate_object_type_prop` For storing types and the associated number of arguments.

```
17 \prop_new:N \g__xtemplate_object_type_prop
(End definition for \g__xtemplate_object_type_prop This variable is documented on page ??.)
```

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

18 `\tl_new:N \l__xtemplate_assignments_tl`
(End definition for \l__xtemplate_assignments_tl This variable is documented on page ??.)

`\l__xtemplate_collection_tl` The current instance collection name is stored here.

19 `\tl_new:N \l__xtemplate_collection_tl`
(End definition for \l__xtemplate_collection_tl This variable is documented on page ??.)

`\l__xtemplate_collections_prop` Lists current collection in force, indexed by object type.

20 `\prop_new:N \l__xtemplate_collections_prop`
(End definition for \l__xtemplate_collections_prop This variable is documented on page ??.)

`\l__xtemplate_default_tl` The default value for a key is recovered here from the property list in which it is stored. The internal implementation of property lists means that this is safe even with un-escaped # tokens.

21 `\tl_new:N \l__xtemplate_default_tl`
`endmacro`

`\l__xtemplate_error_bool` A flag for errors to be carried forward.

22 `\bool_new:N \l__xtemplate_error_bool`

`\l__xtemplate_global_bool` Used to indicate that assignments should be global.

23 `\bool_new:N \l__xtemplate_global_bool`

`\l__xtemplate_restrict_bool` A flag to indicate that a template is being restricted.

24 `\bool_new:N \l__xtemplate_restrict_bool`

`\l__xtemplate_restrict_clist` A scratch list for restricting templates.

25 `\clist_new:N \l__xtemplate_restrict_clist`

`\l__xtemplate_key_name_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.

`\l__xtemplate_keytype_tl`

`\l__xtemplate_keytype_arg_tl` 26 `\tl_new:N \l__xtemplate_key_name_tl`

`\l__xtemplate_value_tl` 27 `\tl_new:N \l__xtemplate_keytype_tl`

`\l__xtemplate_var_tl` 28 `\tl_new:N \l__xtemplate_keytype_arg_tl`

29 `\tl_new:N \l__xtemplate_value_tl`

30 `\tl_new:N \l__xtemplate_var_tl`

`\l__xtemplate_keytypes_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

`\l__xtemplate_key_order_seq`

`\l__xtemplate_values_prop` 31 `\prop_new:N \l__xtemplate_keytypes_prop`

`\l__xtemplate_vars_prop` 32 `\seq_new:N \l__xtemplate_key_order_seq`

33 `\prop_new:N \l__xtemplate_values_prop`

34 `\prop_new:N \l__xtemplate_vars_prop`

<code>\l__xtemplate_tmp_clist</code> <code>\l__xtemplate_tmp_dim</code> <code>\l__xtemplate_tmp_int</code> <code>\l__xtemplate_tmp_muskip</code> <code>\l__xtemplate_tmp_skip</code> <code>\l__xtemplate_tmp_tl</code>	<p>For pre-processing the data stored by <code>xtemplate</code>, a number of scratch variables are needed. The assignments are made to these in the first instance, unless evaluation is delayed.</p> <pre> 35 \clist_new:N \l__xtemplate_tmp_clist 36 \dim_new:N \l__xtemplate_tmp_dim 37 \int_new:N \l__xtemplate_tmp_int 38 \muskip_new:N \l__xtemplate_tmp_muskip 39 \skip_new:N \l__xtemplate_tmp_skip </pre> <p>A scratch variable for comparisons and so on.</p> <pre> 40 \tl_new:N \l__xtemplate_tmp_tl </pre>
---	--

11.2 Variant of prop functions

<code>\prop_get:NoNTF</code>	<p>In some cases, we need to expand the key, and get the corresponding value in a property list if it exists.</p> <pre> 41 \cs_generate_variant:Nn \prop_get:NnNTF { No } 42 \cs_generate_variant:Nn \prop_get:NnNT { No } 43 \cs_generate_variant:Nn \prop_get:NnNF { No } </pre>
------------------------------	--

11.3 Testing existence and validity

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

<code>_xtemplate_execute_if_arg_agree:nnT</code>	<p>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</p> <pre> 44 \cs_new_protected:Npn _xtemplate_execute_if_arg_agree:nnT #1#2#3 45 { 46 \prop_get:NnN \g__xtemplate_object_type_prop {#1} \l__xtemplate_tmp_tl 47 \int_compare:nNnTF {#2} = \l__xtemplate_tmp_tl 48 {#3} 49 { 50 \msg_error:nnxxx { xtemplate } 51 { argument-number-mismatch } {#1} { \l__xtemplate_tmp_tl } {#2} 52 } 53 } </pre>
---	--

<code>_xtemplate_execute_if_code_exist:nnT</code>	<p>A template is only fully declared if the code has been set up, which can be checked by looking for the template function itself.</p> <pre> 54 \cs_new_protected:Npn _xtemplate_execute_if_code_exist:nnT #1#2#3 55 { 56 \cs_if_exist:cTF { \c__xtemplate_code_root_tl #1 / #2 } 57 {#3} 58 { 59 \msg_error:nnxx { xtemplate } { no-template-code } 60 {#1} {#2} </pre>
--	--

```

61     }
62 }

```

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

```

63 \cs_new_protected:Npn \_xtemplate_execute_if_keytype_exist:nT #1#2
64 {
65     \cs_if_exist:CTF { \_xtemplate_store_value_ #1 :n }
66     {#2}
67     { \msg_error:nnx { xtemplate } { unknown-keytype } {#1} }
68 }
69 \cs_generate_variant:Nn \_xtemplate_execute_if_keytype_exist:nT { o }

```

_xtemplate_execute_if_type_exist:nT To check that a particular object type is valid.

```

70 \cs_new_protected:Npn \_xtemplate_execute_if_type_exist:nT #1#2
71 {
72     \prop_if_in:NnTF \g__xtemplate_object_type_prop {#1}
73     {#2}
74     { \msg_error:nnx { xtemplate } { unknown-object-type } {#1} }
75 }

```

_xtemplate_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.

```

76 \cs_new_protected:Npn \_xtemplate_if_keys_exist:nnT #1#2#3
77 {
78     \cs_if_exist:CTF { \c__xtemplate_keytypes_root_tl #1 / #2 }
79     {#3}
80     {
81         \msg_error:nnxx { xtemplate } { unknown-template }
82         {#1} {#2}
83     }
84 }

```

_xtemplate_if_key_value:nT Tests for the first token in a string being \KeyValue, where \EvaluateNow is not impor-
_xtemplate_if_key_value:oT tant.

```

85 \prg_new_conditional:Npnn \_xtemplate_if_key_value:n #1 { T }
86 {
87     \str_if_eq:noTF { \KeyValue } { \tl_head:w #1 \q_nil \q_stop }
88     { \prg_return_true: }
89     { \prg_return_false: }
90 }
91 \cs_generate_variant:Nn \_xtemplate_if_key_value:nT { o }

```

_xtemplate_if_eval_now:nTF Tests for the first token in a string being \EvaluateNow.

```

92 \prg_new_conditional:Npnn \_xtemplate_if_eval_now:n #1 { TF }
93 {
94     \str_if_eq:noTF { \EvaluateNow } { \tl_head:w #1 \q_nil \q_stop }
95     { \prg_return_true: }

```

```

96     { \prg_return_false: }
97 }

```

_xtemplate_if_instance_exist:nnnTF Testing for an instance is collection dependent.

```

98 \prg_new_conditional:Npnn \_xtemplate_if_instance_exist:nnn #1#2#3
99 { T, F, TF }
100 {
101     \cs_if_exist:CTF { \c__xtemplate_instances_root_tl #1 / #2 / #3 }
102     { \prg_return_true: }
103     { \prg_return_false: }
104 }

```

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

```

105 \prg_new_conditional:Npnn \_xtemplate_if_use_template:n #1 { TF }
106 {
107     \str_if_eq:noTF { \UseTemplate } { \tl_head:w #1 \q_nil \q_stop }
108     { \prg_return_true: }
109     { \prg_return_false: }
110 }

```

11.4 Saving and recovering property lists

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

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

```

\_xtemplate_store_defaults:n
\_xtemplate_store_keytypes:n
\_xtemplate_store_restrictions:n
\_xtemplate_store_values:n
\_xtemplate_store_vars:n
111 \cs_new_protected:Npn \_xtemplate_store_defaults:n #1
112 {
113     \prop_gclear_new:c { \c__xtemplate_defaults_root_tl #1 }
114     \prop_gset_eq:cN { \c__xtemplate_defaults_root_tl #1 }
115     \l__xtemplate_values_prop
116 }
117 \cs_new_protected:Npn \_xtemplate_store_keytypes:n #1
118 {
119     \prop_gclear_new:c { \c__xtemplate_keytypes_root_tl #1 }
120     \prop_gset_eq:cN { \c__xtemplate_keytypes_root_tl #1 }
121     \l__xtemplate_keytypes_prop
122     \seq_gclear_new:c { \c__xtemplate_key_order_root_tl #1 }
123     \seq_gset_eq:cN { \c__xtemplate_key_order_root_tl #1 }
124     \l__xtemplate_key_order_seq
125 }
126 \cs_new_protected:Npn \_xtemplate_store_values:n #1
127 {
128     \prop_gclear_new:c { \c__xtemplate_values_root_tl #1 }
129     \prop_gset_eq:cN { \c__xtemplate_values_root_tl #1 }
130     \l__xtemplate_values_prop
131 }
132 \cs_new_protected:Npn \_xtemplate_store_restrictions:n #1
133 {

```

```

134     \clist_gclear_new:c { \c__xtemplate_restrict_root_tl #1 }
135     \clist_gset_eq:cN { \c__xtemplate_restrict_root_tl #1 }
136         \l__xtemplate_restrict_clist
137     }
138 \cs_new_protected:Npn \__xtemplate_store_vars:n #1
139 {
140     \prop_gclear_new:c { \c__xtemplate_vars_root_tl #1 }
141     \prop_gset_eq:cN { \c__xtemplate_vars_root_tl #1 }
142         \l__xtemplate_vars_prop
143 }

```

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.

```

\__xtemplate_recover_defaults:n
\__xtemplate_recover_keytypes:n
\__xtemplate_recover_restrictions:n
\__xtemplate_recover_values:n
\__xtemplate_recover_vars:n
144 \cs_new_protected:Npn \__xtemplate_recover_defaults:n #1
145 {
146     \prop_set_eq:Nc \l__xtemplate_values_prop
147         { \c__xtemplate_defaults_root_tl #1 }
148 }
149 \cs_new_protected:Npn \__xtemplate_recover_keytypes:n #1
150 {
151     \prop_set_eq:Nc \l__xtemplate_keytypes_prop
152         { \c__xtemplate_keytypes_root_tl #1 }
153     \seq_set_eq:Nc \l__xtemplate_key_order_seq
154         { \c__xtemplate_key_order_root_tl #1 }
155 }
156 \cs_new_protected:Npn \__xtemplate_recover_restrictions:n #1
157 {
158     \clist_set_eq:Nc \l__xtemplate_restrict_clist
159         { \c__xtemplate_restrict_root_tl #1 }
160 }
161 \cs_new_protected:Npn \__xtemplate_recover_values:n #1
162 {
163     \prop_set_eq:Nc \l__xtemplate_values_prop
164         { \c__xtemplate_values_root_tl #1 }
165 }
166 \cs_new_protected:Npn \__xtemplate_recover_vars:n #1
167 {
168     \prop_set_eq:Nc \l__xtemplate_vars_prop
169         { \c__xtemplate_vars_root_tl #1 }
170 }

```

11.5 Creating new object types

Although the object 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 object type.

```

171 \cs_new_protected:Npn \__xtemplate_declare_object_type:nn #1#2
172 {
173     \int_set:Nn \l__xtemplate_tmp_int {#2}

```

```

174     \bool_if:nTF
175     {
176         \int_compare_p:nNn {#2} > \c_nine ||
177         \int_compare_p:nNn {#2} < \c_zero
178     }
179     {
180         \msg_error:nnxx { xtemplate } { bad-number-of-arguments }
181         {#1} { \exp_not:V \l__xtemplate_tmp_int }
182     }
183     {
184         \msg_info:nnxx { xtemplate } { declare-object-type }
185         {#1} {#2}
186         \prop_gput:NnV \g__xtemplate_object_type_prop {#1}
187         \l__xtemplate_tmp_int
188     }
189 }

```

11.6 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.

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

```

190 \cs_new_protected:Npn \__xtemplate_declare_template_keys:nnnn #1#2#3#4
191 {
192     \__xtemplate_execute_if_type_exist:nT {#1}
193     {
194         \__xtemplate_execute_if_arg_agree:nnT {#1} {#3}
195         {
196             \prop_clear:N \l__xtemplate_values_prop
197             \prop_clear:N \l__xtemplate_keytypes_prop
198             \seq_clear:N \l__xtemplate_key_order_seq
199             \keyval_parse:NNn
200             \__xtemplate_parse_keys_elt:n \__xtemplate_parse_keys_elt:nn {#4}
201             \__xtemplate_store_defaults:n { #1 / #2 }
202             \__xtemplate_store_keytypes:n { #1 / #2 }
203         }
204     }
205 }

```

`__xtemplate_parse_keys_elt:n` 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.

```

206 \cs_new_protected:Npn \__xtemplate_parse_keys_elt:n #1
207 {
208   \__xtemplate_split_keytype:n {#1}
209   \bool_if:NF \l__xtemplate_error_bool
210   {
211     \__xtemplate_execute_if_keytype_exist:oT \l__xtemplate_keytype_tl
212     {
213       \seq_map_function:NN \c__xtemplate_keytypes_arg_seq
214       \__xtemplate_parse_keys_elt_aux:n
215       \bool_if:NF \l__xtemplate_error_bool
216       {
217         \seq_if_in:NoTF \l__xtemplate_key_order_seq
218         \l__xtemplate_key_name_tl
219         {
220           \msg_error:nxx { xtemplate }
221             { duplicate-key-interface }
222             { \l__xtemplate_key_name_tl }
223         }
224         { \__xtemplate_parse_keys_elt_aux: }
225       }
226     }
227   }
228 }
229 \cs_new_protected_nopar:Npn \__xtemplate_parse_keys_elt_aux:n #1
230 {
231   \str_if_eq:onT \l__xtemplate_keytype_tl {#1}
232   {
233     \tl_if_empty:NT \l__xtemplate_keytype_arg_tl
234     {
235       \msg_error:nxx { xtemplate }
236         { keytype-requires-argument } {#1}
237       \bool_set_true:N \l__xtemplate_error_bool
238       \seq_map_break:
239     }
240   }
241 }
242 \cs_new_nopar:Npn \__xtemplate_parse_keys_elt_aux:
243 {
244   \tl_set:Nx \l__xtemplate_tmp_tl
245   {
246     \l__xtemplate_keytype_tl
247     \tl_if_empty:NF \l__xtemplate_keytype_arg_tl
248     { { \l__xtemplate_keytype_arg_tl } }
249   }
250   \prop_put:Noo \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
251   \l__xtemplate_tmp_tl

```

```

252 \seq_put_right:No \l__xtemplate_key_order_seq \l__xtemplate_key_name_tl
253 \str_if_eq:onT \l__xtemplate_keytype_tl { choice }
254 {
255     \clist_if_in:NnT \l__xtemplate_keytype_arg_tl { unknown }
256     { \msg_error:nn { xtemplate } { choice-unknown-reserved } }
257 }
258 }

```

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

```

259 \cs_new_protected:Npn \__xtemplate_parse_keys_elt:nn #1#2
260 {
261     \__xtemplate_parse_keys_elt:n {#1}
262     \use:c { __xtemplate_store_value_ \l__xtemplate_keytype_tl :n } {#2}
263 }

```

__xtemplate_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, spaces are removed and the category code of colons is standardised. After that, the standard delimited argument method is used to separate the two parts.

__xtemplate_split_keytype_aux:w

```

264 \group_begin:
265 \char_set_lccode:nn { \@ } { \@: }
266 \char_set_catcode_other:N \@
267 \tl_to_lowercase:n
268 {
269     \group_end:
270     \cs_new_protected:Npn \__xtemplate_split_keytype:n #1
271     {
272         \bool_set_false:N \l__xtemplate_error_bool
273         \tl_set:Nn \l__xtemplate_tmp_tl {#1}
274         \tl_remove_all:Nn \l__xtemplate_tmp_tl { ~ }
275         \tl_replace_all:Nnn \l__xtemplate_tmp_tl { : } { @ }
276         \tl_if_in:onTF \l__xtemplate_tmp_tl { @ }
277         {
278             \tl_clear:N \l__xtemplate_key_name_tl
279             \exp_after:wN \__xtemplate_split_keytype_aux:w
280             \l__xtemplate_tmp_tl \q_stop
281         }
282         {
283             \bool_set_true:N \l__xtemplate_error_bool
284             \msg_error:nxx { xtemplate } { missing-keytype } {#1}
285         }
286     }
287     \cs_new_protected:Npn \__xtemplate_split_keytype_aux:w #1 @ #2 \q_stop
288     {
289         \tl_put_right:Nx \l__xtemplate_key_name_tl { \tl_to_str:n {#1} }
290         \tl_if_in:nnTF {#2} { @ }
291         {

```

```

292         \tl_put_right:Nn \l__xtemplate_key_name_tl { @ }
293         \__xtemplate_split_keytype_aux:w #2 \q_stop
294     }
295     {
296         \tl_if_empty:NTF \l__xtemplate_key_name_tl
297         { \msg_error:nxx { xtemplate } { empty-key-name } { @ #2 } }
298         { \__xtemplate_split_keytype_arg:n {#2} }
299     }
300 }
301 }

```

```

\__xtemplate_split_keytype_arg:n
\__xtemplate_split_keytype_arg:o
\__xtemplate_split_keytype_arg_aux:n
\__xtemplate_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).

```

302 \cs_new_protected:Npn \__xtemplate_split_keytype_arg:n #1
303 {
304     \tl_set:Nn \l__xtemplate_keytype_tl {#1}
305     \tl_clear:N \l__xtemplate_keytype_arg_tl
306     \cs_set_protected_nopar:Npn \__xtemplate_split_keytype_arg_aux:n ##1
307     {
308         \tl_if_in:nnT {#1} {###1}
309         {
310             \cs_set:Npn \__xtemplate_split_keytype_arg_aux:w
311             #####1 ##1 #####2 \q_stop
312             {
313                 \tl_if_empty:nT {#####1}
314                 {
315                     \tl_set:Nn \l__xtemplate_keytype_tl {##1}
316                     \tl_set:Nn \l__xtemplate_keytype_arg_tl {#####2}
317                     \seq_map_break:
318                 }
319             }
320             \__xtemplate_split_keytype_arg_aux:w #1 \q_stop
321         }
322     }
323     \seq_map_function:NN \c__xtemplate_keytypes_arg_seq
324     \__xtemplate_split_keytype_arg_aux:n
325 }
326 \cs_generate_variant:Nn \__xtemplate_split_keytype_arg:n { o }
327 \cs_new_nopar:Npn \__xtemplate_split_keytype_arg_aux:n #1 { }
328 \cs_new_nopar:Npn \__xtemplate_split_keytype_arg_aux:w #1 \q_stop { }

```

(End definition for \l__xtemplate_default_tl This function is documented on page ??.)

11.6.1 Storing values

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

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

`__xtemplate_store_value_boolean:n` Storing Boolean values requires a test for delayed evaluation, but is different to the various numerical variable types as there are only two possible values to store. So the code here tests the default switch and then records the meaning (either `true` or `false`).

```

329 \cs_new_protected:Npn \__xtemplate_store_value_boolean:n #1
330 {
331   \__xtemplate_if_eval_now:nTF {#1}
332   {
333     \bool_if:cTF { c_ #1 _bool }
334     {
335       \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl
336       { true }
337     }
338     {
339       \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl
340       { false }
341     }
342   }
343   {
344     \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl {#1}
345   }
346 }
```

(End definition for __xtemplate_store_value_boolean:n This function is documented on page ??.)

`__xtemplate_store_value_code:n` `__xtemplate_store_value_choice:n` `__xtemplate_store_value_commalist:n` `__xtemplate_store_value_function:n` `__xtemplate_store_value_instance:n` `__xtemplate_store_value_real:n` `__xtemplate_store_value_tokenlist:n` With no need to worry about delayed evaluation, these keytypes all just store the input directly.

```

347 \cs_new_protected:Npn \__xtemplate_store_value_code:n #1
348   { \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl {#1} }
349 \cs_new_eq:NN \__xtemplate_store_value_choice:n \__xtemplate_store_value_code:n
350 \cs_new_eq:NN \__xtemplate_store_value_commalist:n \__xtemplate_store_value_code:n
351 \cs_new_eq:NN \__xtemplate_store_value_function:n \__xtemplate_store_value_code:n
352 \cs_new_eq:NN \__xtemplate_store_value_instance:n \__xtemplate_store_value_code:n
353 \cs_new_eq:NN \__xtemplate_store_value_real:n \__xtemplate_store_value_code:n
354 \cs_new_eq:NN \__xtemplate_store_value_tokenlist:n \__xtemplate_store_value_code:n
```

(End definition for __xtemplate_store_value_code:n This function is documented on page ??.)

`_xtemplate_store_value_integer:n`
`_xtemplate_store_value_length:n`
`_xtemplate_store_value_muskip:n`
`_xtemplate_store_value_skip:n`

Storing the value of a number is in all cases more or less the same. If evaluation is taking place now, assignment is made to a scratch variable, and this result is then stored. On the other hand, if evaluation is delayed the current data is simply stored “as is”.

```

355 \cs_new_protected:Npn \_xtemplate_store_value_integer:n #1
356 {
357   \_xtemplate_if_eval_now:nTF {#1}
358   {
359     \int_set:Nn \l__xtemplate_tmp_int {#1}
360     \prop_put:NVV \l__xtemplate_values_prop \l__xtemplate_key_name_int
361       \l__xtemplate_tmp_int
362   }
363   {
364     \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl {#1}
365   }
366 }
367 \cs_new_protected:Npn \_xtemplate_store_value_length:n #1
368 {
369   \_xtemplate_if_eval_now:nTF {#1}
370   {
371     \dim_set:Nn \l__xtemplate_tmp_dim {#1}
372     \prop_put:NVV \l__xtemplate_values_prop \l__xtemplate_key_name_tl
373       \l__xtemplate_tmp_dim
374   }
375   {
376     \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl {#1}
377   }
378 }
379 \cs_new_protected:Npn \_xtemplate_store_value_muskip:n #1
380 {
381   \_xtemplate_if_eval_now:nTF {#1}
382   {
383     \muskip_set:Nn \l__xtemplate_tmp_muskip {#1}
384     \prop_put:NVV \l__xtemplate_values_prop \l__xtemplate_key_name_tl
385       \l__xtemplate_tmp_muskip
386   }
387   {
388     \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl {#1}
389   }
390 }
391 \cs_new_protected:Npn \_xtemplate_store_value_skip:n #1
392 {
393   \_xtemplate_if_eval_now:nTF {#1}
394   {
395     \skip_set:Nn \l__xtemplate_tmp_skip {#1}
396     \prop_put:NVV \l__xtemplate_values_prop \l__xtemplate_key_name_tl
397       \l__xtemplate_tmp_skip
398   }
399   {
400     \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl {#1}

```

```

401     }
402 }

```

(End definition for `__xtemplate_store_value_integer:n` This function is documented on page ??.)

11.7 Implementation part of template declaration

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

```

403 \cs_new_protected:Npn __xtemplate_declare_template_code:nnnnn #1#2#3#4#5
404 {
405   \__xtemplate_execute_if_type_exist:nT {#1}
406   {
407     \__xtemplate_execute_if_arg_agree:nnT {#1}{#3}
408     {
409       \__xtemplate_if_keys_exist:nnT {#1} {#2}
410       {
411         \__xtemplate_store_key_implementation:nnn {#1} {#2} {#4}
412         \cs_generate_from_arg_count:cNnn
413         { \c__xtemplate_code_root_tl #1 / #2 }
414         \cs_gset_protected:Npn {#3} {#5}
415       }
416     }
417   }
418 }

```

(End definition for `__xtemplate_declare_template_code:nnnnn` This function is documented on page ??.)

`__xtemplate_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.

```

419 \cs_new_protected:Npn __xtemplate_store_key_implementation:nnn #1#2#3
420 {
421   \__xtemplate_recover_defaults:n { #1 / #2 }
422   \__xtemplate_recover_keytypes:n { #1 / #2 }
423   \prop_clear:N \l__xtemplate_vars_prop
424   \keyval_parse:NNn
425   \__xtemplate_parse_vars_elt:n \__xtemplate_parse_vars_elt:nn {#3}
426   \__xtemplate_store_vars:n { #1 / #2 }
427   \clist_clear:N \l__xtemplate_restrict_clist
428   \__xtemplate_store_restrictions:n { #1 / #2 }
429   \prop_map_inline:Nn \l__xtemplate_keytypes_prop
430   {
431     \msg_error:nnxxx { xtemplate } { key-not-implemented }
432     {##1} {#2} {#1}
433   }
434 }

```

(End definition for `__xtemplate_store_key_implementation:nnn` This function is documented on page ??.)

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

```
435 \cs_new_protected:Npn \__xtemplate_parse_vars_elt:n #1
436 { \msg_error:nnx { xtemplate } { key-no-variable } {#1} }
```

(End definition for `__xtemplate_parse_vars_elt:n` This function is documented on page ??.)

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

```
437 \cs_new_protected:Npn \__xtemplate_parse_vars_elt:nn #1#2
438 {
439   \tl_set:Nx \l__xtemplate_key_name_tl { \tl_to_str:n {#1} }
440   \tl_remove_all:Nn \l__xtemplate_key_name_tl { ~ }
441   \prop_get:NoNTF
442     \l__xtemplate_keytypes_prop
443     \l__xtemplate_key_name_tl
444     \l__xtemplate_keytype_tl
445     {
446       \__xtemplate_split_keytype_arg:o \l__xtemplate_keytype_tl
447       \__xtemplate_parse_vars_elt_aux:n {#2}
448       \prop_remove:NV \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
449     }
450   { \msg_error:nnx { xtemplate } { unknown-key } {#1} }
451 }
```

(End definition for `__xtemplate_parse_vars_elt:nn` This function is documented on page ??.)

`__xtemplate_parse_vars_elt_aux:n` There now needs to be some sanity checking on the variable name given. This does not
`__xtemplate_parse_vars_elt_aux:w` apply for choice or code “variables”, but in all other cases the variable needs to exist. Also, the only prefix acceptable is global. So there are a few related checks to make.

```
452 \cs_new_protected:Npn \__xtemplate_parse_vars_elt_aux:n #1
453 {
454   \str_if_eq:onTF \l__xtemplate_keytype_tl { choice }
455   { \__xtemplate_implement_choices:n {#1} }
456   {
457     \str_if_eq:onTF \l__xtemplate_keytype_tl { code }
458     {
459       \prop_put:Non \l__xtemplate_vars_prop
460       \l__xtemplate_key_name_tl {#1}
461     }
462     {
463       \tl_if_single:nTF {#1}
464       {
465         \cs_if_exist:NF #1
466         { \__xtemplate_create_variable:N #1 }
467         \prop_put:Non \l__xtemplate_vars_prop
468         \l__xtemplate_key_name_tl {#1}

```

```

469     }
470     {
471         \tl_if_in:nnTF {#1} { global }
472         { \__xtemplate_parse_vars_elt_aux:w #1 \q_stop }
473         {
474             \msg_error:nnx { xtemplate } { bad-variable }
475             { \tl_to_str:n {#1} }
476         }
477     }
478 }
479 }
480 }
481 \cs_new_protected:Npn \__xtemplate_parse_vars_elt_aux:w #1 global #2 \q_stop
482 {
483     \tl_if_empty:nTF {#1}
484     {
485         \tl_if_single:nTF {#2}
486         {
487             \cs_if_exist:NF #2
488             { \__xtemplate_create_variable:N #2 }
489             \prop_put:Non \l__xtemplate_vars_prop
490             \l__xtemplate_key_name_tl { #1 global #2 }
491         }
492         {
493             \msg_error:nnx { xtemplate } { bad-variable }
494             { \tl_to_str:n { #1 global #2 } }
495         }
496     }
497     {
498         \msg_error:nnx { xtemplate } { bad-variable }
499         { \tl_to_str:n { #1 global #2 } }
500     }
501 }

```

(End definition for __xtemplate_parse_vars_elt_aux:n This function is documented on page ??.)

`__xtemplate_create_variable:N` A shortcut to create non-declared variables. Some types need a name mapping, others can be used directly.

```

502 \cs_new_protected_nopar:Npn \__xtemplate_create_variable:N #1
503 {
504     \str_case:onnn \l__xtemplate_keytype_tl
505     {
506         { boolean } { \bool_new:N #1 }
507         { commalist } { \clist_new:N #1 }
508         { function } { \cs_new:Npn #1 { } }
509         { instance } { \cs_new_protected:Npn #1 { } }
510         { integer } { \int_new:N #1 }
511         { length } { \dim_new:N #1 }
512         { real } { \fp_new:N #1 }
513         { tokenlist } { \tl_new:N #1 }

```



```

514     }
515     { \use:c { \l__xtemplate_keytype_tl _ new:N } #1 }
516 }

```

(End definition for __xtemplate_create_variable:N This function is documented on page ??.)

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

```

\__xtemplate_implement_choices_default:
517 \cs_new_protected:Npn \__xtemplate_implement_choices:n #1
518 {
519   \clist_set_eq:NN \l__xtemplate_tmp_clist \l__xtemplate_keytype_arg_tl
520   \prop_put:Non \l__xtemplate_vars_prop \l__xtemplate_key_name_tl { }
521   \keyval_parse:NNn
522     \__xtemplate_implement_choice_elt:n \__xtemplate_implement_choice_elt:nn
523     {#1}
524   \prop_get:NoNT \l__xtemplate_values_prop \l__xtemplate_key_name_tl
525     \l__xtemplate_tmp_tl
526   { \__xtemplate_implement_choices_default: }
527   \clist_if_empty:NF \l__xtemplate_tmp_clist
528   {
529     \clist_map_inline:Nn \l__xtemplate_tmp_clist
530     {
531       \msg_error:nnx { xtemplate } { choice-not-implemented }
532       {##1}
533     }
534   }
535 }

```

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

```

536 \cs_new_protected_nopar:Npn \__xtemplate_implement_choices_default:
537 {
538   \tl_set:Nx \l__xtemplate_tmp_tl
539     { \l__xtemplate_key_name_tl \c_space_tl \l__xtemplate_tmp_tl }
540   \prop_if_in:NoF \l__xtemplate_vars_prop \l__xtemplate_tmp_tl
541   {
542     \tl_set:Nx \l__xtemplate_tmp_tl
543       { \l__xtemplate_key_name_tl \c_space_tl \l__xtemplate_tmp_tl }
544     \prop_if_in:NoF \l__xtemplate_vars_prop \l__xtemplate_tmp_tl
545     {
546       \prop_get:NoN \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
547         \l__xtemplate_tmp_tl
548       \__xtemplate_split_keytype_arg:o \l__xtemplate_tmp_tl
549       \prop_get:NoN \l__xtemplate_values_prop \l__xtemplate_key_name_tl
550         \l__xtemplate_tmp_tl
551       \msg_error:nnxxx { xtemplate } { unknown-default-choice }
552       { \l__xtemplate_key_name_tl } { \l__xtemplate_key_name_tl }
553       { \l__xtemplate_keytype_arg_tl }
554     }
555   }
556 }

```

(End definition for `_xtemplate_implement_choices:n` This function is documented on page ??.)

`_xtemplate_implement_choice_elt:n` The actual storage of the implementation of a choice is mainly about error checking. The
`_xtemplate_implement_choice_elt:nn` 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.

```

557 \cs_new_protected:Npn \_xtemplate_implement_choice_elt:n #1
558 {
559   \clist_if_empty:NTF \l__xtemplate_tmp_clist
560   {
561     \str_if_eq:nnF {#1} { unknown }
562     {
563       \prop_get:NoN \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
564       \l__xtemplate_tmp_tl
565       \_xtemplate_split_keytype_arg:o \l__xtemplate_tmp_tl
566       \msg_error:nnxxx { xtemplate } { unknown-choice }
567       { \l__xtemplate_key_name_tl } {#1}
568       { \l__xtemplate_keytype_arg_tl }
569     }
570   }
571   {
572     \clist_if_in:NnTF \l__xtemplate_tmp_clist {#1}
573     { \clist_remove_all:Nn \l__xtemplate_tmp_clist {#1} }
574     {
575       \prop_get:NoN \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
576       \l__xtemplate_tmp_tl
577       \_xtemplate_split_keytype_arg:o \l__xtemplate_tmp_tl
578       \msg_error:nnxxx { xtemplate } { unknown-choice }
579       { \l__xtemplate_key_name_tl } {#1}
580       { \l__xtemplate_keytype_arg_tl }
581     }
582   }
583 }
584 \cs_new_protected:Npn \_xtemplate_implement_choice_elt:nn #1#2
585 {
586   \_xtemplate_implement_choice_elt:n {#1}
587   \tl_set:Nx \l__xtemplate_tmp_tl
588   { \l__xtemplate_key_name_tl \c_space_tl #1 }
589   \prop_put:Non \l__xtemplate_vars_prop \l__xtemplate_tmp_tl {#2}
590 }

```

(End definition for `_xtemplate_implement_choice_elt:n` This function is documented on page ??.)

11.8 Editing template defaults

Template defaults can be edited either with no other changes or to prevent further editing, forming a “restricted template”. In the later case, a new template results, whereas simple editing does not produce a new template name.

`_xtemplate_declare_restricted:nnnn` Creating a restricted template means copying the old template to the new one first.

```

591 \cs_new_protected:Npn \_xtemplate_declare_restricted:nnnn #1#2#3#4
592 {
593   \_xtemplate_if_keys_exist:nnT {#1} {#2}
594   {
595     \_xtemplate_set_template_eq:nn { #1 / #3 } { #1 / #2 }
596     \bool_set_true:N \l__xtemplate_restrict_bool
597     \_xtemplate_edit_defaults_aux:nnn {#1} {#3} {#4}
598   }
599 }

```

(End definition for `_xtemplate_declare_restricted:nnnn` This function is documented on page ??.)

`_xtemplate_edit_defaults:nnn` Editing the template defaults means getting the values back out of the store, then parsing
`_xtemplate_edit_defaults_aux:nnn` the list of new values before putting the updated list back into storage. The auxiliary
function is used to allow code-sharing with the template-restriction system.

```

600 \cs_new_protected:Npn \_xtemplate_edit_defaults:nnn
601 {
602   \bool_set_false:N \l__xtemplate_restrict_bool
603   \_xtemplate_edit_defaults_aux:nnn
604 }
605 \cs_new_protected:Npn \_xtemplate_edit_defaults_aux:nnn #1#2#3
606 {
607   \_xtemplate_if_keys_exist:nnT {#1} {#2}
608   {
609     \_xtemplate_recover_defaults:n { #1 / #2 }
610     \_xtemplate_recover_restrictions:n { #1 / #2 }
611     \_xtemplate_parse_values:nn { #1 / #2 } {#3}
612     \_xtemplate_store_defaults:n { #1 / #2 }
613     \_xtemplate_store_restrictions:n { #1 / #2 }
614   }
615 }

```

(End definition for `_xtemplate_edit_defaults:nnn` This function is documented on page ??.)

`_xtemplate_parse_values:nn` 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.

```

616 \cs_new_protected:Npn \_xtemplate_parse_values:nn #1#2
617 {
618   \_xtemplate_recover_keytypes:n {#1}
619   \clist_clear:N \l__xtemplate_restrict_clist
620   \keyval_parse:NNn
621   \_xtemplate_parse_values_elt:n \_xtemplate_parse_values_elt:nn {#2}
622 }

```

(End definition for `_xtemplate_parse_values:nn` This function is documented on page ??.)

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

```

623 \cs_new_protected:Npn \_xtemplate_parse_values_elt:n #1
624 {
625   \bool_set_true:N \l__xtemplate_error_bool

```

```

626     \msg_error:nnx { xtemplate } { key-no-value } {#1}
627 }

```

(End definition for _xtemplate_parse_values_elt:n This function is documented on page ??.)

_xtemplate_parse_values_elt:nn To store the value, find the keytype then call the saving function. These need the current key name saved as \l__xtemplate_key_name_tl. When a template is being restricted, the setting code will be skipped for restricted keys.

```

628 \cs_new_protected:Npn \_xtemplate_parse_values_elt:nn #1#2
629 {
630     \tl_set:Nx \l__xtemplate_key_name_tl { \tl_to_str:n {#1} }
631     \tl_remove_all:Nn \l__xtemplate_key_name_tl { ~ }
632     \prop_get:NoNTF \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
633     \l__xtemplate_tmp_tl
634     {
635         \bool_if:NTF \l__xtemplate_restrict_bool
636         {
637             \clist_if_in:NoF \l__xtemplate_restrict_clist
638             \l__xtemplate_key_name_tl
639             { \_xtemplate_parse_values_elt_aux:n {#2} }
640         }
641         { \_xtemplate_parse_values_elt_aux:n {#2} }
642     }
643     {
644         \msg_error:nnx { xtemplate } { unknown-key }
645         { \l__xtemplate_key_name_tl }
646     }
647 }
648 \cs_new_protected:Npn \_xtemplate_parse_values_elt_aux:n #1
649 {
650     \clist_put_right:No \l__xtemplate_restrict_clist \l__xtemplate_key_name_tl
651     \_xtemplate_split_keytype_arg:o \l__xtemplate_tmp_tl
652     \use:c { __xtemplate_store_value_ \l__xtemplate_keytype_tl :n } {#1}
653 }

```

(End definition for _xtemplate_parse_values_elt:nn This function is documented on page ??.)

_xtemplate_set_template_eq:nn 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.

```

654 \cs_new_protected:Npn \_xtemplate_set_template_eq:nn #1#2
655 {
656     \_xtemplate_recover_defaults:n {#2}
657     \_xtemplate_store_defaults:n {#1}
658     \_xtemplate_recover_keytypes:n {#2}
659     \_xtemplate_store_keytypes:n {#1}
660     \_xtemplate_recover_vars:n {#2}
661     \_xtemplate_store_vars:n {#1}
662     \cs_gset_eq:cc { \c__xtemplate_code_root_tl #1 }
663     { \c__xtemplate_code_root_tl #2 }
664 }

```

(End definition for _xtemplate_set_template_eq:nn This function is documented on page ??.)

11.9 Creating instances of templates

_xtemplate_declare_instance:nnnn
_xtemplate_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. A final check is also made so that there is always an instance “outside” of any collection.

```

665 \cs_new_protected:Npn \_xtemplate_declare_instance:nnnn #1#2#3#4#5
666 {
667   \_xtemplate_execute_if_code_exist:nnT {#1} {#2}
668   {
669     \_xtemplate_recover_defaults:n { #1 / #2 }
670     \_xtemplate_recover_vars:n { #1 / #2 }
671     \_xtemplate_declare_instance_aux:nnnn {#1} {#2} {#3} {#4} {#5}
672   }
673 }
674 \cs_new_protected:Npn \_xtemplate_declare_instance_aux:nnnn #1#2#3#4#5
675 {
676   \bool_set_false:N \l__xtemplate_error_bool
677   \_xtemplate_parse_values:nn { #1 / #2 } {#5}
678   \bool_if:NF \l__xtemplate_error_bool
679   {
680     \prop_put:Nnn \l__xtemplate_values_prop { from~template } {#2}
681     \_xtemplate_store_values:n { #1 / #3 / #4 }
682     \_xtemplate_convert_to_assignments:
683     \cs_set_protected:cpx { \c__xtemplate_instances_root_tl #1 / #3 / #4 }
684     {
685       \exp_not:N \_xtemplate_assignments_push:n
686       { \exp_not:o \l__xtemplate_assignments_tl }
687       \exp_not:c { \c__xtemplate_code_root_tl #1 / #2 }
688     }
689     \_xtemplate_if_instance_exist:nnnF {#1} { } {#4}
690     {
691       \cs_set_eq:cc
692       { \c__xtemplate_instances_root_tl #1 /      / #4 }
693       { \c__xtemplate_instances_root_tl #1 / #3 / #4 }
694     }
695   }
696 }

```

(End definition for _xtemplate_declare_instance:nnnn This function is documented on page ??.)

_xtemplate_edit_instance:nnnn
_xtemplate_edit_instance_aux:nnnn
_xtemplate_edit_instance_aux:nnnn

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.

```

697 \cs_new_protected:Npn \_xtemplate_edit_instance:nnnn #1#2#3
698 {
699   \_xtemplate_if_instance_exist:nnnTF {#1} {#2} {#3}
700   {
701     \_xtemplate_recover_values:n { #1 / #2 / #3 }

```

```

702     \prop_get:NnN \l__xtemplate_values_prop { from-template }
703     \l__xtemplate_tmp_tl
704     \__xtemplate_edit_instance_aux:nonnn {#1} \l__xtemplate_tmp_tl
705     {#2} {#3}
706   }
707   {
708     \msg_error:nxxx { xtemplate } { unknown-instance }
709     {#1} {#3}
710   }
711 }
712 \cs_new_protected:Npn \__xtemplate_edit_instance_aux:nnnnn #1#2
713 {
714   \__xtemplate_recover_vars:n { #1 / #2 }
715   \__xtemplate_declare_instance_aux:nnnnn {#1} {#2}
716 }
717 \cs_generate_variant:Nn \__xtemplate_edit_instance_aux:nnnnn { no }

```

(End definition for __xtemplate_edit_instance:nnnn This function is documented on page ??.)

```

\__xtemplate_convert_to_assignments:
\__xtemplate_convert_to_assignments_aux:n
\__xtemplate_convert_to_assignments_aux:nn
\__xtemplate_convert_to_assignments_aux:no

```

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”.

```

718 \cs_new_protected_nopar:Npn \__xtemplate_convert_to_assignments:
719 {
720   \tl_clear:N \l__xtemplate_assignments_tl
721   \seq_map_function:NN \l__xtemplate_key_order_seq
722   \__xtemplate_convert_to_assignments_aux:n
723 }
724 \cs_new_protected:Npn \__xtemplate_convert_to_assignments_aux:n #1
725 {
726   \prop_get:NnN \l__xtemplate_keytypes_prop {#1} \l__xtemplate_tmp_tl
727   \__xtemplate_convert_to_assignments_aux:no {#1} \l__xtemplate_tmp_tl
728 }

```

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.

```

729 \cs_new_protected:Npn \__xtemplate_convert_to_assignments_aux:nn #1#2
730 {
731   \prop_get:NnNT \l__xtemplate_values_prop {#1} \l__xtemplate_value_tl
732   {
733     \prop_get:NnNTF \l__xtemplate_vars_prop {#1} \l__xtemplate_var_tl
734     {
735       \__xtemplate_split_keytype_arg:n {#2}
736       \str_if_eq:onF \l__xtemplate_keytype_tl { choice }
737       {
738         \str_if_eq:onF \l__xtemplate_keytype_tl { code }
739         { \__xtemplate_find_global: }

```

```

740     }
741     \tl_set:Nn \l__xtemplate_key_name_tl {#1}
742     \use:c { __xtemplate_assign_ \l__xtemplate_keytype_tl : }
743   }
744   { \msg_error:nnx { xtemplate } { unknown-attribute } {#1} }
745 }
746 }
747 \cs_generate_variant:Nn \__xtemplate_convert_to_assignments_aux:nn { no }
(End definition for \__xtemplate_convert_to_assignments: This function is documented on page ??.)

```

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

```

748 \cs_new_protected_nopar:Npn \__xtemplate_find_global:
749 {
750   \bool_set_false:N \l__xtemplate_global_bool
751   \tl_if_in:onT \l__xtemplate_var_tl { global }
752   {
753     \exp_after:wN \__xtemplate_find_global_aux:w \l__xtemplate_var_tl \q_stop
754   }
755 }
756 \cs_new_protected:Npn \__xtemplate_find_global_aux:w #1 global #2 \q_stop
757 {
758   \tl_set:Nn \l__xtemplate_var_tl {#2}
759   \bool_set_true:N \l__xtemplate_global_bool
760 }
(End definition for \__xtemplate_find_global: This function is documented on page ??.)

```

11.10 Using templates directly

`__xtemplate_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.

```

761 \cs_new_protected:Npn \__xtemplate_use_template:nnn #1#2#3
762 {
763   \__xtemplate_execute_if_code_exist:nnT {#1} {#2}
764   {
765     \__xtemplate_recover_defaults:n { #1 / #2 }
766     \__xtemplate_recover_vars:n { #1 / #2 }
767     \__xtemplate_parse_values:nn { #1 / #2 } {#3}
768     \__xtemplate_convert_to_assignments:
769     \use:c { \c__xtemplate_code_root_tl #1 / #2 }
770   }
771 }
(End definition for \__xtemplate_use_template:nnn This function is documented on page ??.)

```

11.11 Assigning values to variables

`__xtemplate_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.

```

772 \cs_new_protected_nopar:Npn \__xtemplate_assign_boolean:
773 {
774   \bool_if:NTF \l__xtemplate_global_bool
775     { \__xtemplate_assign_boolean_aux:n { bool_gset } }
776     { \__xtemplate_assign_boolean_aux:n { bool_set } }
777 }
778 \cs_new_protected_nopar:Npn \__xtemplate_assign_boolean_aux:n #1
779 {
780   \__xtemplate_if_key_value:oT \l__xtemplate_value_tl
781     { \__xtemplate_key_to_value: }
782   \tl_put_left:Nx \l__xtemplate_assignments_tl
783     {
784       \exp_not:c { #1 _ \l__xtemplate_value_tl :N }
785       \exp_not:o \l__xtemplate_var_tl
786     }
787 }
```

(End definition for `__xtemplate_assign_boolean:` This function is documented on page ??.)

`__xtemplate_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.

```

788 \cs_new_protected_nopar:Npn \__xtemplate_assign_choice:
789 {
790   \__xtemplate_assign_choice_aux:xF
791   { \l__xtemplate_key_name_tl \c_space_tl \l__xtemplate_value_tl }
792   {
793     \__xtemplate_assign_choice_aux:xF
794     { \l__xtemplate_key_name_tl \c_space_tl unknown }
795     {
796       \prop_get:NoN \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
797         \l__xtemplate_tmp_tl
798       \__xtemplate_split_keytype_arg:o \l__xtemplate_tmp_tl
799       \msg_error:nnxxx { xtemplate } { unknown-choice }
800       { \l__xtemplate_key_name_tl } { \l__xtemplate_value_tl }
801       { \l__xtemplate_keytype_arg_tl }
802     }
803   }
804 }
805 \cs_new_protected_nopar:Npn \__xtemplate_assign_choice_aux:nF #1
806 {
807   \prop_get:NnNTF
808     \l__xtemplate_vars_prop
809     {#1}
810     \l__xtemplate_tmp_tl
811     { \tl_put_right:No \l__xtemplate_assignments_tl \l__xtemplate_tmp_tl }
```



```

812 }
813 \cs_generate_variant:Nn \__xtemplate_assign_choice_aux:nF { x }
(End definition for \__xtemplate_assign_choice: This function is documented on page ??.)

```

__xtemplate_assign_code: Assigning general code to a key needs a scratch function to be created and run when
__xtemplate_assign_code:n \AssignTemplateKeys is called. So the appropriate definition then use is created in the
token list variable.

```

814 \cs_new_protected_nopar:Npn \__xtemplate_assign_code:
815 {
816   \tl_put_left:Nx \l__xtemplate_assignments_tl
817   {
818     \cs_set_protected:Npn \__xtemplate_assign_code:n \exp_not:n {##1}
819     { \exp_not:o \l__xtemplate_var_tl }
820     \__xtemplate_assign_code:n { \exp_not:o \l__xtemplate_value_tl }
821   }
822 }
823 \cs_new_protected:Npn \__xtemplate_assign_code:n #1 { }
(End definition for \__xtemplate_assign_code: This function is documented on page ??.)

```

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

```

\__xtemplate_assign_function_aux:N
824 \cs_new_protected_nopar:Npn \__xtemplate_assign_function:
825 {
826   \bool_if:NTF \l__xtemplate_global_bool
827   { \__xtemplate_assign_function_aux:N \cs_gset:Npn }
828   { \__xtemplate_assign_function_aux:N \cs_set:Npn }
829 }
830 \cs_new_protected_nopar:Npn \__xtemplate_assign_function_aux:N #1
831 {
832   \tl_put_left:Nx \l__xtemplate_assignments_tl
833   {
834     \cs_generate_from_arg_count:NNnn
835     \exp_not:o \l__xtemplate_var_tl
836     \exp_not:N #1
837     { \exp_not:o \l__xtemplate_keytype_arg_tl }
838     { \exp_not:o \l__xtemplate_value_tl }
839   }
840 }
(End definition for \__xtemplate_assign_function: This function is documented on page ??.)

```

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

```

841 \cs_new_protected_nopar:Npn \__xtemplate_assign_instance:
842 {
843   \bool_if:NTF \l__xtemplate_global_bool
844   { \__xtemplate_assign_instance_aux:N \cs_gset_protected:Npn }
845   { \__xtemplate_assign_instance_aux:N \cs_set_protected:Npn }
846 }
847 \cs_new_protected_nopar:Npn \__xtemplate_assign_instance_aux:N #1
848 {

```

```

849 \tl_put_left:Nx \l__xtemplate_assignments_tl
850 {
851   \exp_not:N #1 \exp_not:o \l__xtemplate_var_tl
852   {
853     \__xtemplate_use_instance:nn
854     { \exp_not:o \l__xtemplate_keytype_arg_tl }
855     { \exp_not:o \l__xtemplate_value_tl }
856   }
857 }
858 }

```

(End definition for __xtemplate_assign_instance: This function is documented on page ??.)

__xtemplate_assign_integer: All of the calculated assignments use the same underlying code, with only the low-level assignment function changing.

```

\__xtemplate_assign_length:
\__xtemplate_assign_muskip:
\__xtemplate_assign_real:
\__xtemplate_assign_skip:
859 \cs_new_protected_nopar:Npn \__xtemplate_assign_integer:
860 {
861   \bool_if:NTF \l__xtemplate_global_bool
862   { \__xtemplate_assign_variable:N \int_gset:Nn }
863   { \__xtemplate_assign_variable:N \int_set:Nn }
864 }
865 \cs_new_protected_nopar:Npn \__xtemplate_assign_length:
866 {
867   \bool_if:NTF \l__xtemplate_global_bool
868   { \__xtemplate_assign_variable:N \dim_gset:Nn }
869   { \__xtemplate_assign_variable:N \dim_set:Nn }
870 }
871 \cs_new_protected_nopar:Npn \__xtemplate_assign_muskip:
872 {
873   \bool_if:NTF \l__xtemplate_global_bool
874   { \__xtemplate_assign_variable:N \muskip_gset:Nn }
875   { \__xtemplate_assign_variable:N \muskip_set:Nn }
876 }
877 \cs_new_protected_nopar:Npn \__xtemplate_assign_real:
878 {
879   \bool_if:NTF \l__xtemplate_global_bool
880   { \__xtemplate_assign_variable:N \fp_gset:Nn }
881   { \__xtemplate_assign_variable:N \fp_set:Nn }
882 }
883 \cs_new_protected_nopar:Npn \__xtemplate_assign_skip:
884 {
885   \bool_if:NTF \l__xtemplate_global_bool
886   { \__xtemplate_assign_variable:N \skip_gset:Nn }
887   { \__xtemplate_assign_variable:N \skip_set:Nn }
888 }

```

(End definition for __xtemplate_assign_integer: This function is documented on page ??.)

__xtemplate_assign_tokenlist: Storing lists of tokens is easy: no complex calculations and no need to worry about numbers of arguments.

```

\__xtemplate_assign_tokenlist_aux:N
889 \cs_new_protected_nopar:Npn \__xtemplate_assign_tokenlist:

```

```

890 {
891     \bool_if:NTF \l__xtemplate_global_bool
892     { \__xtemplate_assign_tokenlist_aux:N \tl_gset:Nn }
893     { \__xtemplate_assign_tokenlist_aux:N \tl_set:Nn }
894 }
895 \cs_new_protected_nopar:Npn \__xtemplate_assign_tokenlist_aux:N #1
896 {
897     \__xtemplate_if_key_value:oT \l__xtemplate_value_tl
898     { \__xtemplate_key_to_value: }
899     \tl_put_left:Nx \l__xtemplate_assignments_tl
900     {
901         #1 \exp_not:o \l__xtemplate_var_tl
902         { \exp_not:o \l__xtemplate_value_tl }
903     }
904 }

```

(End definition for __xtemplate_assign_tokenlist: This function is documented on page ??.)

__xtemplate_assign_commalist: Very similar for commas lists, so some code is shared.

```

905 \cs_new_protected_nopar:Npn \__xtemplate_assign_commalist:
906 {
907     \bool_if:NTF \l__xtemplate_global_bool
908     { \__xtemplate_assign_tokenlist_aux:N \clist_gset:Nn }
909     { \__xtemplate_assign_tokenlist_aux:N \clist_set:Nn }
910 }

```

(End definition for __xtemplate_assign_commalist: This function is documented on page ??.)

__xtemplate_assign_variable:N A general-purpose function for all of the numerical assignments. As long as the value is not coming from another variable, the stored value is simply transferred for output.

```

911 \cs_new_protected_nopar:Npn \__xtemplate_assign_variable:N #1
912 {
913     \__xtemplate_if_key_value:oT \l__xtemplate_value_tl
914     { \__xtemplate_key_to_value: }
915     \tl_put_left:Nx \l__xtemplate_assignments_tl
916     {
917         #1 \exp_not:o \l__xtemplate_var_tl
918         { \exp_not:o \l__xtemplate_value_tl }
919     }
920 }

```

(End definition for __xtemplate_assign_variable:N This function is documented on page ??.)

__xtemplate_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.

__xtemplate_key_to_value_aux:w

```

921 \cs_new_protected_nopar:Npn \__xtemplate_key_to_value:
922 { \exp_after:wN \__xtemplate_key_to_value_aux:w \l__xtemplate_value_tl }
923 \cs_new_protected:Npn \__xtemplate_key_to_value_aux:w \KeyValue #1
924 {
925     \tl_set:Nx \l__xtemplate_tmp_tl { \tl_to_str:n {#1} }

```

```

926 \tl_remove_all:Nn \l__xtemplate_key_name_tl { ~ }
927 \prop_get:NoNF
928 \l__xtemplate_vars_prop
929 \l__xtemplate_tmp_tl
930 \l__xtemplate_value_tl
931 {
932   \msg_error:nxx { xtemplate } { unknown-attribute }
933   { \l__xtemplate_tmp_tl }
934 }
935 }

```

(End definition for __xtemplate_key_to_value: This function is documented on page ??.)

11.12 Using instances

__xtemplate_use_instance:nn
 __xtemplate_use_instance_aux:nNnnn
 __xtemplate_use_instance_aux:nn

Using an instance is just a question of finding the appropriate function. There is the possibility that a collection instance exists, so this is checked before trying the general instance. If nothing is found, an error is raised. One additional 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.

```

936 \cs_new_protected:Npn \__xtemplate_use_instance:nn #1#2
937 {
938   \__xtemplate_if_use_template:nTF {#2}
939   { \__xtemplate_use_instance_aux:nNnnn {#1} #2 }
940   { \__xtemplate_use_instance_aux:nn {#1} {#2} }
941 }
942 \cs_new_protected:Npn \__xtemplate_use_instance_aux:nNnnn #1#2#3#4#5
943 {
944   \str_if_eq:nnTF {#1} {#3}
945   { \__xtemplate_use_template:nnn {#3} {#4} {#5} }
946   { \msg_error:nxx { xtemplate } { type-mismatch } {#1} {#3} }
947 }
948 \cs_new_protected:Npn \__xtemplate_use_instance_aux:nn #1#2
949 {
950   \__xtemplate_get_collection:n {#1}
951   \__xtemplate_if_instance_exist:nnnTF
952   {#1} { \l__xtemplate_collection_tl } {#2}
953   {
954     \use:c
955     {
956       \c__xtemplate_instances_root_tl #1 /
957       \l__xtemplate_collection_tl / #2
958     }
959   }
960   {
961     \__xtemplate_if_instance_exist:nnnTF {#1} { } {#2}
962     { \use:c { \c__xtemplate_instances_root_tl #1 / / #2 } }
963     {
964       \msg_error:nxx { xtemplate } { unknown-instance }
965       {#1} {#2}

```

```

966         }
967     }
968 }
(End definition for \__xtemplate_use_instance:nn This function is documented on page ??.)

```

__xtemplate_use_collection:nn Switching to an instance collection is just a question of setting the appropriate list.

```

969 \cs_new_protected:Npn \__xtemplate_use_collection:nn #1#2
970 { \prop_put:Nnn \l__xtemplate_collections_prop {#1} {#2} }
(End definition for \__xtemplate_use_collection:nn This function is documented on page ??.)

```

__xtemplate_get_collection:n Recovering the collection for a given type is pretty easy: just a read from the list.

```

971 \cs_new_protected:Npn \__xtemplate_get_collection:n #1
972 {
973     \prop_get:NnNF \l__xtemplate_collections_prop {#1}
974     \l__xtemplate_collection_tl
975     { \tl_clear:N \l__xtemplate_collection_tl }
976 }
(End definition for \__xtemplate_get_collection:n This function is documented on page ??.)

```

11.13 Assignment manipulation

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

__xtemplate_assignments_pop: To actually use the assignments.

```

977 \cs_new_nopar:Npn \__xtemplate_assignments_pop: { \l__xtemplate_assignments_tl }
(End definition for \__xtemplate_assignments_pop: This function is documented on page ??.)

```

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

```

978 \cs_new_protected:Npn \__xtemplate_assignments_push:n #1
979 { \tl_set:Nn \l__xtemplate_assignments_tl {#1} }
(End definition for \__xtemplate_assignments_push:n This function is documented on page ??.)

```

11.14 Showing templates and instances

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

```

980 \cs_new_protected_nopar:Npn \__xtemplate_show_code:nn #1#2
981 { \cs_show:c { \c__xtemplate_code_root_tl #1 / #2 } }
(End definition for \__xtemplate_show_code:nn This function is documented on page ??.)

```

__xtemplate_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.

__xtemplate_show_keytypes:nn

__xtemplate_show_vars:nn

__xtemplate_show:Nnnn

```

982 \cs_new_protected_nopar:Npn \__xtemplate_show_defaults:nn #1#2
983 {
984     \__xtemplate_if_keys_exist:nnT {#1} {#2}
985     {
986         \__xtemplate_recover_defaults:n { #1 / #2 }
987         \__xtemplate_show:Nnnn \l__xtemplate_values_prop
988         {#1} {#2} { default~values }

```

```

989     }
990   }
991   \cs_new_protected_nopar:Npn \__xtemplate_show_keytypes:nn #1#2
992   {
993     \__xtemplate_if_keys_exist:nnT {#1} {#2}
994     {
995       \__xtemplate_recover_keytypes:n { #1 / #2 }
996       \__xtemplate_show:Nnnn \l__xtemplate_keytypes_prop
997       {#1} {#2} { interface }
998     }
999   }
1000   \cs_new_protected_nopar:Npn \__xtemplate_show_vars:nn #1#2
1001   {
1002     \__xtemplate_execute_if_code_exist:nnT {#1} {#2}
1003     {
1004       \__xtemplate_recover_vars:n { #1 / #2 }
1005       \__xtemplate_show:Nnnn \l__xtemplate_vars_prop
1006       {#1} {#2} { variable~mapping }
1007     }
1008   }
1009   \cs_new_protected_nopar:Npn \__xtemplate_show:Nnnn #1#2#3#4
1010   {
1011     \__msg_term:nnnnn { xtemplate }
1012     { \prop_if_empty:NTF #1 { show-no-attribute } { show-attribute } }
1013     {#2} {#3} {#4}
1014     \__msg_show_variable:x
1015     { \prop_map_function:NN #1 \__msg_show_item_unbraced:nn }
1016   }

```

(End definition for __xtemplate_show_defaults:nn, __xtemplate_show_keytypes:nn, and __xtemplate_show_vars:nn
 These functions are documented on page ??.)

\template_show_values:nnn Instance values are a little more complex, as there are the collection and template to consider.

```

1017   \cs_new_protected_nopar:Npn \__xtemplate_show_values:nnn #1#2#3
1018   {
1019     \__xtemplate_if_instance_exist:nnnT {#1} {#2} {#3}
1020     {
1021       \__xtemplate_recover_values:n { #1 / #2 / #3 }
1022       \prop_if_empty:NTF \l__xtemplate_values_prop
1023       {
1024         \__msg_term:nnnnn { xtemplate } { show-no-values }
1025         {#1} {#2} {#3}
1026         \__msg_show_variable:x { }
1027       }
1028       {
1029         \prop_pop:NnN \l__xtemplate_values_prop { from~template }
1030         \l__xtemplate_tmp_tl
1031         \__msg_term:nnnnnV { xtemplate } { show-values }
1032         {#1} {#2} {#3} \l__xtemplate_tmp_tl
1033         \__msg_show_variable:x

```

```

1034         {
1035             \prop_map_function:NN \l__xtemplate_values_prop
1036             \__msg_show_item_unbraced:nn
1037         }
1038     }
1039 }
1040 }

```

(End definition for \template_show_values:nnn This function is documented on page ??.)

11.15 Messages

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

```

1041 \msg_new:nnnn { xtemplate } { argument-number-mismatch }
1042 { Object~type~'#1'~takes~#2~argument(s). }
1043 {
1044     \c_msg_coding_error_text_tl
1045     Objects~of~type~'#1'~require~#2~argument(s).\
1046     You~have~tried~to~make~a~template~for~'#1'~
1047     with~#3~argument(s),~which~is~not~possible:~
1048     the~number~of~arguments~must~agree.
1049 }
1050 \msg_new:nnnn { xtemplate } { bad-number-of-arguments }
1051 { Bad~number~of~arguments~for~object~type~'#1'. }
1052 {
1053     \c_msg_coding_error_text_tl
1054     An~object~may~accept~between~0~and~9~arguments.\
1055     You~asked~to~use~#2~arguments:~this~is~not~supported.
1056 }
1057 \msg_new:nnnn { xtemplate } { bad-variable }
1058 { Incorrect~variable~description~'#1'. }
1059 {
1060     The~argument~'#1'~is~not~of~the~form \
1061     ~'<variable>' \
1062     ~or~ \
1063     ~'global~<variable>' \
1064     It~must~be~given~in~one~of~these~formats~to~be~used~in~a~template.
1065 }
1066 \msg_new:nnnn { xtemplate } { choice-not-implemented }
1067 { The~choice~'#1'~has~no~implementation. }
1068 {
1069     Each~choice~listed~in~the~interface~for~a~template~must~
1070     have~an~implementation.
1071 }
1072 \msg_new:nnnn { xtemplate } { choice-no-code }
1073 { The~choice~'#1'~requires~implementation~details. }
1074 {
1075     \c_msg_coding_error_text_tl
1076     When~creating~template~code~using~\DeclareTemplateCode,~
1077     each~choice~name~must~have~an~associated~implementation.\

```

```

1078     This~should~be~given~after~a~'='~sign:~LaTeX~did~not~find~one.
1079 }
1080 \msg_new:nnnn { xtemplate } { duplicate-key-interface }
1081 { Key~'#1'~appears~twice~in~interface~definition~\msg_line_context:. }
1082 {
1083     \c_msg_coding_error_text_tl
1084     Each~key~can~only~have~one~interface~declared~in~a~template.\\
1085     LaTeX~found~two~interfaces~for~'#1'.
1086 }
1087 \msg_new:nnnn { xtemplate } { keytype-requires-argument }
1088 { The~key~type~'#1'~requires~an~argument~\msg_line_context:. }
1089 {
1090     You~should~have~put:\\
1091     \ \ <key-name>~'#1'~{~<argument>~} \ \
1092     but~LaTeX~did~not~find~an~<argument>.
1093 }
1094 \msg_new:nnnn { xtemplate } { invalid-keytype }
1095 { The~key~'#1'~is~missing~a~key~type~\msg_line_context:. }
1096 {
1097     \c_msg_coding_error_text_tl
1098     Each~key~in~a~template~requires~a~key~type,~given~in~the~form:\\
1099     \ \ <key>~::~~<key-type>\\
1100     LaTeX~could~not~find~a~<key-type>~in~your~input.
1101 }
1102 \msg_new:nnnn { xtemplate } { key-no-value }
1103 { The~key~'#1'~has~no~value~\msg_line_context:. }
1104 {
1105     \c_msg_coding_error_text_tl
1106     When~creating~an~instance~of~a~template~
1107     every~key~listed~must~include~a~value:\\
1108     \ \ <key>~::~~<value>
1109 }
1110 \msg_new:nnnn { xtemplate } { key-no-variable }
1111 { The~key~'#1'~requires~implementation~details~\msg_line_context:. }
1112 {
1113     \c_msg_coding_error_text_tl
1114     When~creating~template~code~using~\DeclareTemplateCode,~
1115     each~key~name~must~have~an~associated~implementation.\\
1116     This~should~be~given~after~a~'='~sign:~LaTeX~did~not~find~one.
1117 }
1118 \msg_new:nnnn { xtemplate } { key-not-implemented }
1119 { Key~'#1'~has~no~implementation~\msg_line_context:. }
1120 {
1121     \c_msg_coding_error_text_tl
1122     The~definition~of~key~implementations~for~template~'#2'~
1123     of~object~type~'#3'~does~not~include~any~details~for~key~'#1'.\\
1124     The~key~was~declared~in~the~interface~definition,~
1125     and~so~an~implementation~is~required.
1126 }
1127 \msg_new:nnnn { xtemplate } { missing-keytype }

```



```

1128 { The-key~'#1'~is missing~a~key-type~\msg_line_context:. }
1129 {
1130   \c_msg_coding_error_text_tl
1131   Key~interface~definitions~should~be~of~the~form\\
1132   \ \ #1~::~~<key-type>\\
1133   but~LaTeX~could~not~find~a~<key-type>.
1134 }
1135 \msg_new:nnnn { xtemplate } { no-template-code }
1136 {
1137   The~template~'#2'~of~type~'#1'~is~unknown~
1138   or~has~no~implementation.
1139 }
1140 {
1141   \c_msg_coding_error_text_tl
1142   There~is~no~code~available~for~the~template~name~given.\\
1143   This~should~be~given~using~\DeclareTemplateCode.
1144 }
1145 \msg_new:nnnn { xtemplate } { object-type-mismatch }
1146 { Object~types~'#1'~and~'#2'~do~not~agree. }
1147 {
1148   You~are~trying~to~use~a~template~directly~with~\UseInstance
1149   (or~a~similar~function),~but~the~object~types~do~not~match.
1150 }
1151 \msg_new:nnnn { xtemplate } { unknown-attribute }
1152 { The~template~attribute~'#1'~is~unknown. }
1153 {
1154   There~is~a~definition~in~the~current~template~reading\\
1155   \ \ \token_to_str:N \KeyValue {~#1~} \ \
1156   but~there~is~no~key~called~'#1'.
1157 }
1158 \msg_new:nnnn { xtemplate } { unknown-choice }
1159 { The~choice~'#2'~was~not~declared~for~key~'#1'. }
1160 {
1161   The~key~'#1'~takes~a~fixed~list~of~choices~
1162   and~this~list~does~not~include~'#2'.
1163 }
1164 \msg_new:nnnn { xtemplate } { unknown-default-choice }
1165 { The~default~choice~'#2'~was~not~declared~for~key~'#1'. }
1166 {
1167   The~key~'#1'~takes~a~fixed~list~of~choices~
1168   and~this~list~does~not~include~'#2'.
1169 }
1170 \msg_new:nnnn { xtemplate } { unknown-instance }
1171 { The~instance~'#2'~of~type~'#1'~is~unknown. }
1172 {
1173   You~have~asked~to~use~an~instance~'#2',~
1174   but~this~has~not~been~created.
1175 }
1176 \msg_new:nnnn { xtemplate } { unknown-key }
1177 { Unknown~template~key~'#1'. }

```

```

1178 {
1179     \c_msg_coding_error_text_t1
1180     The~key~'#1'~was~not~declared~in~the~interface~
1181     for~the~current~template.
1182 }
1183 \msg_new:nnnn { xtemplate } { unknown-keytype }
1184 { The~key~type~'#1'~is~unknown. }
1185 {
1186     \c_msg_coding_error_text_t1
1187     Valid~key~types~are:\\
1188     --boolean;\\
1189     --choice;\\
1190     --code;\\
1191     --commalist;\\
1192     --function;\\
1193     --instance;\\
1194     --integer;\\
1195     --length;\\
1196     --muskip;\\
1197     --real;\\
1198     --skip;\\
1199     --tokenlist.
1200 }
1201 \msg_new:nnnn { xtemplate } { unknown-object-type }
1202 { The~object~type~'#1'~is~unknown. }
1203 {
1204     \c_msg_coding_error_text_t1
1205     An~object~type~needs~to~be~declared~with~\DeclareObjectType
1206     prior~to~using~it.
1207 }
1208 \msg_new:nnnn { xtemplate } { unknown-template }
1209 { The~template~'#2'~of~type~'#1'~is~unknown. }
1210 {
1211     No~interface~has~been~declared~for~a~template~
1212     '#2'~of~object~type~'#1'.
1213 }

```

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

```

1214 \msg_new:nnn { xtemplate } { declare-object-type }
1215 { Declaring~object~type~'#1'~taking~#2~argument(s)~\msg_line_context:. }
1216 \msg_new:nnn { xtemplate } { declare-template-code }
1217 { Declaring~code~for~template~'#2'~of~object~type~'#1'~\msg_line_context:. }
1218 \msg_new:nnn { xtemplate } { declare-template-interface }
1219 {
1220     Declaring~interface~for~template~'#2'~of~object~type~'#1'~
1221     \msg_line_context:.
1222 }
1223 \msg_new:nnn { xtemplate } { show-no-attribute }
1224 { The~template~'#2'~of~object~type~'#1'~has~no~#3 . }
1225 \msg_new:nnn { xtemplate } { show-attribute }

```

```

1226 { The~template~'#2'~of-object-type~'#1'~has~#3 : }
1227 \msg_new:nnn { xtemplate } { show-no-values }
1228 {
1229   The~ \tl_if_empty:nF {#2} {collection~} instance~'#3'~
1230   \tl_if_empty:nF {#2} { (from~collection~'#2')~ }
1231   of-object-type~'#1'~has~no~values.
1232 }
1233 \msg_new:nnn { xtemplate } { show-values }
1234 {
1235   The~ \tl_if_empty:nF {#2} {collection~} instance~'#3'~
1236   \tl_if_empty:nF {#2} { (from~collection~'#2')~ }
1237   of-object-type~'#1'~
1238   \str_if_eq:nnF { \q_no_value } {#4} { (from~template~'#4')~ }
1239   has~values:
1240 }

```

11.16 User functions

The user functions provided by xtemplate are pretty much direct copies of internal ones. However, by sticking to the xparse approach only the appropriate arguments are long.

<pre> \DeclareObjectType \DeclareTemplateInterface \DeclareTemplateCode \DeclareRestrictedTemplate \EditTemplateDefaults \DeclareInstance \DeclareCollectionInstance \EditInstance \EditCollectionInstance \UseTemplate \UseInstance \UseCollection </pre>	<p>All simple translations, with the appropriate long/short argument filtering.</p> <pre> 1241 \cs_new_protected_nopar:Npn \DeclareObjectType #1#2 1242 { __xtemplate_declare_object_type:nn {#1} {#2} } 1243 \cs_new_protected:Npn \DeclareTemplateInterface #1#2#3#4 1244 { __xtemplate_declare_template_keys:nnnn {#1} {#2} {#3} {#4} } 1245 \cs_new_protected:Npn \DeclareTemplateCode #1#2#3#4#5 1246 { __xtemplate_declare_template_code:nnnnn {#1} {#2} {#3} {#4} {#5} } 1247 \cs_new_protected:Npn \DeclareRestrictedTemplate #1#2#3#4 1248 { __xtemplate_declare_restricted:nnnn {#1} {#2} {#3} {#4} } 1249 \cs_new_protected:Npn \DeclareInstance #1#2#3#4 1250 { __xtemplate_declare_instance:nnnnn {#1} {#3} { } {#2} {#4} } 1251 \cs_new_protected:Npn \DeclareCollectionInstance #1#2#3#4#5 1252 { __xtemplate_declare_instance:nnnnn {#2} {#4} {#1} {#3} {#5} } 1253 \cs_new_protected:Npn \EditTemplateDefaults #1#2#3 1254 { __xtemplate_edit_defaults:nnn {#1} {#2} {#3} } 1255 \cs_new_protected:Npn \EditInstance #1#2#3 1256 { __xtemplate_edit_instance:nnnn {#1} { } {#2} {#3} } 1257 \cs_new_protected:Npn \EditCollectionInstance #1#2#3#4 1258 { __xtemplate_edit_instance:nnnn {#2} {#1} {#3} {#4} } 1259 \cs_new_protected_nopar:Npn \UseTemplate #1#2#3 1260 { __xtemplate_use_template:nnn {#1} {#2} {#3} } 1261 \cs_new_protected_nopar:Npn \UseInstance #1#2 1262 { __xtemplate_use_instance:nn {#1} {#2} } 1263 \cs_new_protected_nopar:Npn \UseCollection #1#2 1264 { __xtemplate_use_collection:nn {#1} {#2} } </pre>
--	--

(End definition for \DeclareObjectType This function is documented on page ??.)

```

\ShowTemplateCode
\ShowTemplateDefaults
\ShowTemplateInterface
\ShowTemplateVariables
\ShowInstanceValues
\ShowCollectionInstanceValues

```

The show functions are again just translation.

```

1265 \cs_new_protected_nopar:Npn \ShowTemplateCode #1#2
1266   { \__xtemplate_show_code:nn {#1} {#2} }
1267 \cs_new_protected_nopar:Npn \ShowTemplateDefaults #1#2
1268   { \__xtemplate_show_defaults:nn {#1} {#2} }
1269 \cs_new_protected_nopar:Npn \ShowTemplateInterface #1#2
1270   { \__xtemplate_show_keytypes:nn {#1} {#2} }
1271 \cs_new_protected_nopar:Npn \ShowTemplateVariables #1#2
1272   { \__xtemplate_show_vars:nn {#1} {#2} }
1273 \cs_new_protected_nopar:Npn \ShowInstanceValues #1#2
1274   { \__xtemplate_show_values:nnn {#1} { } {#2} }
1275 \cs_new_protected_nopar:Npn \ShowCollectionInstanceValues #1#2#3
1276   { \__xtemplate_show_values:nnn {#1} {#2} {#3} }
(End definition for \ShowTemplateCode This function is documented on page 9.)

```

`\IfInstanceExistTF` More direct translation: only the base instance is checked for.

```

1277 \cs_new_nopar:Npn \IfInstanceExistTF #1#2
1278   { \__xtemplate_if_instance_exist:nnnTF {#1} { } {#2} }
1279 \cs_new_nopar:Npn \IfInstanceExistT #1#2
1280   { \__xtemplate_if_instance_exist:nnnT {#1} { } {#2} }
1281 \cs_new_nopar:Npn \IfInstanceExistF #1#2
1282   { \__xtemplate_if_instance_exist:nnnF {#1} { } {#2} }
(End definition for \IfInstanceExistTF This function is documented on page ??.)

```

`\EvaluateNow` These are both do nothing functions. Both simply dump their arguments when executed:
`\KeyValue` this should not happen with `\KeyValue`.

```

1283 \cs_new_protected:Npn \EvaluateNow #1 {#1}
1284 \cs_new_protected:Npn \KeyValue #1 {#1}
(End definition for \EvaluateNow This function is documented on page 4.)

```

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

```

1285 \cs_new_protected_nopar:Npn \AssignTemplateKeys
1286   { \__xtemplate_assignments_pop: }
(End definition for \AssignTemplateKeys This function is documented on page 5.)

1287 \cs_new_eq:NN \ShowTemplateKeytypes \ShowTemplateInterface
1288 </package>

```

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> 265	1060, 1061, 1062, 1063, 1077, 1084, 1090, 1091, 1098, 1099, 1107, 1115,
<code>\@</code> 265, 266	1123, 1131, 1132, 1142, 1154, 1155,
<code>\%</code> 1045, 1054,	1187, 1188, 1189, 1190, 1191, 1192,

1193, 1194, 1195, 1196, 1197, 1198	_xtemplate_declare_instance:nnnnn
_msg_show_item_unbraced:nn 1015, 1036 665, 665, 1250, 1252
_msg_show_variable:x . 1014, 1026, 1033	_xtemplate_declare_instance_aux:nnnnn
_msg_term:nnnnn 1011, 1024 665, 671, 674, 715
_msg_term:nnnnnV 1031	_xtemplate_declare_object_type:nn
_xtemplate_assign_boolean: .. 772, 772 171, 171, 1242
_xtemplate_assign_boolean_aux:n ..	_xtemplate_declare_restricted:nnnn
..... 772, 775, 776, 778 591, 591, 1248
_xtemplate_assign_choice: ... 788, 788	_xtemplate_declare_template_code:nnnnn
_xtemplate_assign_choice_aux:n .. 788 403, 403, 1246
_xtemplate_assign_choice_aux:nF ..	_xtemplate_declare_template_keys:nnnn
..... 805, 813 190, 190, 1244
_xtemplate_assign_choice_aux:o .. 788	_xtemplate_edit_defaults:nnn
_xtemplate_assign_choice_aux:xF 600, 600, 1254
..... 790, 793	_xtemplate_edit_defaults_aux:nnn .
_xtemplate_assign_code: 814, 814 597, 600, 603, 605
_xtemplate_assign_code:n 814, 818, 820, 823	_xtemplate_edit_instance:nnnn 697, 697, 1256, 1258
_xtemplate_assign_commalist: 905, 905	_xtemplate_edit_instance_aux:nnnnn
_xtemplate_assign_function: . 824, 824 697, 712, 717
_xtemplate_assign_function_aux:N .	_xtemplate_edit_instance_aux:nonnn
..... 824, 827, 828, 830 697, 704
_xtemplate_assign_instance: . 841, 841	_xtemplate_execute_if_arg_agree:nnT
_xtemplate_assign_instance_aux:N 44, 44, 194, 407
..... 841, 844, 845, 847	_xtemplate_execute_if_code_exist:nnT
_xtemplate_assign_integer: .. 859, 859 54, 54, 667, 763, 1002
_xtemplate_assign_length: ... 859, 865	_xtemplate_execute_if_keys_exist:nnT
_xtemplate_assign_muskip: ... 859, 871 76
_xtemplate_assign_real: 859, 877	_xtemplate_execute_if_keytype_exist:nT
_xtemplate_assign_skip: 859, 883 63, 63, 69
_xtemplate_assign_tokenlist: 889, 889	_xtemplate_execute_if_keytype_exist:oT
_xtemplate_assign_tokenlist_aux:N 63, 211
..... 889, 892, 893, 895, 908, 909	_xtemplate_execute_if_type_exist:nT
_xtemplate_assign_variable:N 70, 70, 192, 405
..... 862, 863, 868, 869,	_xtemplate_find_global: . 739, 748, 748
874, 875, 880, 881, 886, 887, 911, 911	_xtemplate_find_global_aux:w
_xtemplate_assignments_pop: 748, 753, 756
..... 977, 977, 1286	_xtemplate_get_collection:n 950, 971, 971
_xtemplate_assignments_push:n 92
..... 685, 978, 978	_xtemplate_if_eval_now:nTF 92, 331, 357, 369, 381, 393
_xtemplate_convert_to_assignments:	_xtemplate_if_instance_exist:nnn . 98
..... 682, 718, 718, 768	_xtemplate_if_instance_exist:nnnF
_xtemplate_convert_to_assignments_aux:n 689, 1282
..... 718, 722, 724	_xtemplate_if_instance_exist:nnnT
_xtemplate_convert_to_assignments_aux:nn 1019, 1280
..... 718, 729, 747	_xtemplate_if_instance_exist:nnnTF
_xtemplate_convert_to_assignments_aux:no 98, 699, 951, 961, 1278
..... 718, 727	_xtemplate_if_key_value:n 85
_xtemplate_create_variable:N	
..... 466, 488, 502, 502	

_xtemplate_if_key_value:nT ...	85, 91	_xtemplate_recover_values:n	144, 161, 701, 1021
_xtemplate_if_key_value:oT	85, 780, 897, 913	_xtemplate_recover_vars:n	144, 166, 660, 670, 714, 766, 1004
_xtemplate_if_keys_exist:nnT	76, 409, 593, 607, 984, 993	_xtemplate_set_template_eq:nn	595, 654, 654
_xtemplate_if_use_template:n ...	105	_xtemplate_show:Nnnn	982, 987, 996, 1005, 1009
_xtemplate_if_use_template:nTF ...	105, 938	_xtemplate_show_code:nn	980, 980, 1266
_xtemplate_implement_choice_elt:n	522, 557, 557, 586	_xtemplate_show_defaults:nn	982, 982, 1268
_xtemplate_implement_choice_elt:nn	522, 557, 584	_xtemplate_show_keytypes:nn	982, 991, 1270
_xtemplate_implement_choices:n ...	455, 517, 517	_xtemplate_show_values:nnn	1017, 1274, 1276
_xtemplate_implement_choices_default:	517, 526, 536	_xtemplate_show_vars:nn	982, 1000, 1272
_xtemplate_key_to_value:	781, 898, 914, 921, 921	_xtemplate_split_keytype:n	208, 264, 270
_xtemplate_key_to_value_aux:w	921, 922, 923	_xtemplate_split_keytype_arg:n ...	298, 302, 302, 326, 735
_xtemplate_parse_keys_elt:n	200, 206, 206, 261	_xtemplate_split_keytype_arg:o ...	302, 446, 548, 565, 577, 651, 798
_xtemplate_parse_keys_elt:nn	200, 259, 259	_xtemplate_split_keytype_arg_aux:n	302, 306, 324, 327
_xtemplate_parse_keys_elt_aux: ...	206, 224, 242	_xtemplate_split_keytype_arg_aux:w	302, 310, 320, 328
_xtemplate_parse_keys_elt_aux:n ..	206, 214, 229	_xtemplate_split_keytype_aux:w ...	264, 279, 287, 293
_xtemplate_parse_values:nn	611, 616, 616, 677, 767	_xtemplate_store_defaults:n	111, 111, 201, 612, 657
_xtemplate_parse_values_elt:n	621, 623, 623	_xtemplate_store_key_implementation:nnn	411, 419, 419
_xtemplate_parse_values_elt:nn ...	621, 628, 628	_xtemplate_store_keytypes:n	111, 117, 202, 659
_xtemplate_parse_values_elt_aux:n	628, 639, 641, 648	_xtemplate_store_restrictions:n ..	111, 132, 428, 613
_xtemplate_parse_vars_elt:n	425, 435, 435	_xtemplate_store_value_boolean:n .	329, 329
_xtemplate_parse_vars_elt:nn	425, 437, 437	_xtemplate_store_value_choice:n ..	347, 349
_xtemplate_parse_vars_elt_aux:n ..	447, 452, 452	_xtemplate_store_value_code:n	347, 347, 349, 350, 351, 352, 353, 354
_xtemplate_parse_vars_elt_aux:w ..	452, 472, 481	_xtemplate_store_value_commalist:n	347, 350
_xtemplate_recover_defaults:n	144, 144, 421, 609, 656, 669, 765, 986	_xtemplate_store_value_function:n	347, 351
_xtemplate_recover_keytypes:n	144, 149, 422, 618, 658, 995	_xtemplate_store_value_instance:n	347, 352
_xtemplate_recover_restrictions:n	144, 156, 610	_xtemplate_store_value_integer:n .	355, 355

_xtemplate_store_value_length:n ..	_c_xtemplate_keytypes_root_tl
..... 355, 367 5, 8, 78, 119, 120, 152
_xtemplate_store_value_muskip:n ..	_c_xtemplate_restrict_root_tl
..... 355, 379 5, 10, 134, 135, 159
_xtemplate_store_value_real:n 347, 353	_c_xtemplate_values_root_tl
_xtemplate_store_value_skip:n 355, 391 5, 11, 128, 129, 164
_xtemplate_store_value_tokenlist:n	_c_xtemplate_vars_root_tl
..... 347, 354 5, 12, 140, 141, 169
_xtemplate_store_values:n 111, 126, 681	_c_msg_coding_error_text_tl .. 1044,
_xtemplate_store_vars:n	1053, 1075, 1083, 1097, 1105, 1113,
..... 111, 138, 426, 661	1121, 1130, 1141, 1179, 1186, 1204
_xtemplate_use_collection:nn	_c_nine
..... 969, 969, 1264 176
_xtemplate_use_instance:nn	_c_space_tl
..... 853, 936, 936, 1262 539, 543, 588, 791, 794
_xtemplate_use_instance_aux:nNnnn	_c_zero
..... 936, 939, 942 177
_xtemplate_use_instance_aux:nn ...	_char_set_catcode_other:N
..... 936, 940, 948 266
_xtemplate_use_template:nnn	_char_set_lccode:nn
..... 761, 761, 945, 1260 265
_	_clist_clear:N
1091, 1099, 1108, 1132, 1155 427, 619
A	_clist_gclear_new:c
\AssignTemplateKeys 134
5, 1285, 1285	_clist_gset:Nn
B 908
\bool_if:cTF	_clist_gset_eq:cN
..... 333 135
\bool_if:NF	_clist_if_empty:NF
..... 209, 215, 678 527
\bool_if:NTF	_clist_if_empty:NnTF
..... 635, 774, 826, 255
843, 861, 867, 873, 879, 885, 891, 907	_clist_if_in:NnTF
\bool_if:nTF 572
..... 174	_clist_if_in:NoF
\bool_new:N 637
..... 22, 23, 24, 506	_clist_map_inline:Nn
\bool_set_false:N 529
..... 272, 602, 676, 750	_clist_new:N
\bool_set_true:N 25, 35, 507
..... 237, 283, 596, 625, 759	_clist_put_right:No
C 650
_c_xtemplate_code_root_tl	_clist_remove_all:Nn
5, 56, 413, 662, 663, 687, 769, 981 573
_c_xtemplate_defaults_root_tl	_clist_set:Nn
..... 5, 6, 113, 114, 147 909
_c_xtemplate_instances_root_tl	_clist_set_eq:Nc
..... 5, 7, 101, 683, 692, 693, 956, 962 158
_c_xtemplate_key_order_root_tl	_clist_set_eq:NN
..... 5, 9, 122, 123, 154 519
_c_xtemplate_keytypes_arg_seq	_cs_generate_from_arg_count:cNnn ..
..... 13, 13, 14, 15, 16, 213, 323 412
	_cs_generate_from_arg_count:NNnn ..
 834
	_cs_generate_variant:Nn
 41, 42, 43, 69, 91, 326, 717, 747, 813
	_cs_gset:Npn
 827
	_cs_gset_eq:cc
 662
	_cs_gset_protected:Npn
 414, 844
	_cs_if_exist:cTF
 56, 65, 78, 101
	_cs_if_exist:NF
 465, 487
	_cs_new:Npn
 508
	_cs_new_eq:NN
 349, 350, 351, 352, 353, 354, 1287
	_cs_new_nopar:Npn
 242, 327, 328, 977, 1277, 1279, 1281
	_cs_new_protected:Npn
 44, 54, 63,
	70, 76, 111, 117, 126, 132, 138, 144,
	149, 156, 161, 166, 171, 190, 206,
	259, 270, 287, 302, 329, 347, 355,

367, 379, 391, 403, 419, 435, 437, 452, 481, 509, 517, 557, 584, 591, 600, 605, 616, 623, 628, 648, 654, 665, 674, 697, 712, 724, 729, 756, 761, 823, 923, 936, 942, 948, 969, 971, 978, 1243, 1245, 1247, 1249, 1251, 1253, 1255, 1257, 1283, 1284	
\cs_new_protected_nopar:Npn ... 229, 502, 536, 718, 748, 772, 778, 788, 805, 814, 824, 830, 841, 847, 859, 865, 871, 877, 883, 889, 895, 905, 911, 921, 980, 982, 991, 1000, 1009, 1017, 1241, 1259, 1261, 1263, 1265, 1267, 1269, 1271, 1273, 1275, 1285	
\cs_set:Npn 310, 828	
\cs_set_eq:cc 691	
\cs_set_protected:cpx 683	
\cs_set_protected:Npn 818, 845	
\cs_set_protected_nopar:Npn 306	
\cs_show:c 981	
D	
\DeclareCollectionInstance .. 1241, 1251	
\DeclareInstance 7, 1241, 1249	
\DeclareObjectType ... 3, 1205, 1241, 1241	
\DeclareRestrictedTemplate 9, 1241, 1247	
\DeclareTemplateCode 5, 1076, 1114, 1143, 1241, 1245	
\DeclareTemplateInterface . 3, 1241, 1243	
\dim_gset:Nn 868	
\dim_new:N 36, 511	
\dim_set:Nn 371, 869	
E	
\EditCollectionInstance 1241, 1257	
\EditInstance 8, 1241, 1255	
\EditTemplateDefaults 8, 1241, 1253	
\EvaluateNow 6, 94, 1283, 1283	
\exp_after:wN 279, 753, 922	
\exp_not:c 687, 784	
\exp_not:N 685, 836, 851	
\exp_not:n 818	
\exp_not:o 686, 785, 819, 820, 835, 837, 838, 851, 854, 855, 901, 902, 917, 918	
\exp_not:V 181	
\ExplFileDate 4	
\ExplFileDescription 4	
\ExplFileName 4	
\ExplFileVersion 4	
F	
\fp_gset:Nn 880	
\fp_new:N 512	
\fp_set:Nn 881	
G	
\g__xtemplate_object_type_prop 17, 17, 46, 72, 186	
\group_begin: 264	
\group_end: 269	
I	
\IfInstanceExistF 1281	
\IfInstanceExistT 1279	
\IfInstanceExistTF 1277, 1277	
\int_compare:nNnTF 47	
\int_compare_p:nNn 176, 177	
\int_gset:Nn 862	
\int_new:N 37, 510	
\int_set:Nn 173, 359, 863	
K	
\keyval_parse:NNn 199, 424, 521, 620	
\KeyValue 4, 87, 923, 1155, 1283, 1284	
L	
\l__xtemplate_assignments_tl 18, 18, 686, 720, 782, 811, 816, 832, 849, 899, 915, 977, 979	
\l__xtemplate_collection_tl 19, 19, 952, 957, 974, 975	
\l__xtemplate_collections_prop 20, 20, 970, 973	
\l__xtemplate_default_tl 21, 21	
\l__xtemplate_error_bool 22, 22, 209, 215, 237, 272, 283, 625, 676, 678	
\l__xtemplate_global_bool 23, 23, 750, 759, 774, 826, 843, 861, 867, 873, 879, 885, 891, 907	
\l__xtemplate_key_name_int 360	
\l__xtemplate_key_name_tl ... 26, 26, 218, 222, 250, 252, 278, 289, 292, 296, 335, 339, 344, 348, 364, 372, 376, 384, 388, 396, 400, 439, 440, 443, 448, 460, 468, 490, 520, 524, 539, 543, 546, 549, 552, 563, 567, 575, 579, 588, 630, 631, 632, 638, 645, 650, 741, 791, 794, 796, 800, 926	
\l__xtemplate_key_order_seq 31, 32, 124, 153, 198, 217, 252, 721	

