

The `xtemplate` package

Prototype document functions*

The L^AT_EX3 Project[†]

Released 2013/07/12

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 v4544, last revised 2013/07/12.

[†]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.

\IfInstanceExistTF

```
\IfInstanceExistTF {\langle object type \rangle} {\langle instance \rangle} {\langle true code \rangle} {\langle false code \rangle}
```

Tests if the named $\langle instance \rangle$ of a $\langle object type \rangle$ exists, and then inserts the appropriate code into the input stream.

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.

`\UseInstance`

`\UseInstance`
`{⟨object type⟩} {⟨instance⟩} ⟨arguments⟩`

Uses an `⟨instance⟩` of the `⟨object type⟩`, which will require `⟨arguments⟩` as determined by the number specified for the `⟨object type⟩`. The `⟨instance⟩` must have been declared before it can be used, otherwise an error is raised.

`\UseTemplate`

`\UseTemplate {⟨object type⟩} {⟨template⟩}`
`{⟨settings⟩} ⟨arguments⟩`

Uses the `⟨template⟩` of the specified `⟨object type⟩`, applying the `⟨settings⟩` and absorbing `⟨arguments⟩` as detailed by the `⟨object type⟩` declaration. This in effect is the same as creating an instance using `\DeclareInstance` and immediately using it with `\UseInstance`, but without the instance having any further existence. 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.

`\EditTemplateDefaults`

`\EditTemplateDefaults`
`{⟨object type⟩} {⟨template⟩} {⟨new defaults⟩}`

Edits the `⟨defaults⟩` for a `⟨template⟩` for an `⟨object type⟩`. The `⟨new defaults⟩`, given as a key-value list, replace the existing defaults for the `⟨template⟩`. This means that the change will apply to instances declared after the editing, but that instances which have already been created are unaffected.

\EditInstance**\EditInstance**
{⟨object type⟩} {⟨instance⟩} {⟨new values⟩}

Edits the *⟨values⟩* for an *⟨instance⟩* for an *⟨object type⟩*. The *⟨new values⟩*, given as a key-value list, replace the existing values for the *⟨instance⟩*. 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.

\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 Collections

The implementation of templates includes a concept termed “collections”. The idea is that by activating a collection, a set of instances can rapidly be set up. An example use case would be collections for `frontmatter`, `mainmatter` and `backmatter` in a book. This mechanism is currently implemented by the commands `\DeclareCollectionInstance`, `\EditCollectionInstance` and `\UseCollection`. However, while the idea of switchable instances is a useful one, the team feel that collections are not the correct way to achieve this, at least with the current approach. As such, the collection functions should be regarded as deprecated: they remain available to support existing code, but will be removed when a better mechanism is developed.

`\ShowCollectionInstanceValues`

`\ShowInstanceValues {<collection>} {<object type>} {<instance>}`

Shows the *<values>* for an *<instance>* within a *<collection>* of the given *<object type>* at the terminal. As for other collection commands, this should be regarded as deprecated.

12 xtemplate Implementation

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

12.1 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
```

So that literal values are kept to a minimum.

```
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

```

<code>\l__xtemplate_key_name_tl</code>	When defining each key in a template, the name and type of the key need to be separated
<code>\l__xtemplate_keytype_tl</code>	and stored. Any argument needed by the keytype is also stored separately.
<code>\l__xtemplate_keytype_arg_tl</code>	<code>26 \tl_new:N \l__xtemplate_key_name_tl</code>
<code>\l__xtemplate_value_tl</code>	<code>27 \tl_new:N \l__xtemplate_keytype_tl</code>
<code>\l__xtemplate_var_tl</code>	<code>28 \tl_new:N \l__xtemplate_keytype_arg_tl</code>
	<code>29 \tl_new:N \l__xtemplate_value_tl</code>
	<code>30 \tl_new:N \l__xtemplate_var_tl</code>
<code>\l__xtemplate_keytypes_prop</code>	To avoid needing too many difficult-to-follow csname assignments, various scratch token
<code>\l__xtemplate_key_order_seq</code>	registers are used to build up data, which is then transferred
<code>\l__xtemplate_values_prop</code>	<code>31 \prop_new:N \l__xtemplate_keytypes_prop</code>
<code>\l__xtemplate_vars_prop</code>	<code>32 \seq_new:N \l__xtemplate_key_order_seq</code>
	<code>33 \prop_new:N \l__xtemplate_values_prop</code>
	<code>34 \prop_new:N \l__xtemplate_vars_prop</code>
<code>\l__xtemplate_tmp_clist</code>	For pre-processing the data stored by xtemplate, a number of scratch variables are needed.
<code>\l__xtemplate_tmp_dim</code>	The assignments are made to these in the first instance, unless evaluation is delayed.
<code>\l__xtemplate_tmp_int</code>	<code>35 \clist_new:N \l__xtemplate_tmp_clist</code>
<code>\l__xtemplate_tmp_muskip</code>	<code>36 \dim_new:N \l__xtemplate_tmp_dim</code>
<code>\l__xtemplate_tmp_skip</code>	<code>37 \int_new:N \l__xtemplate_tmp_int</code>
	<code>38 \muskip_new:N \l__xtemplate_tmp_muskip</code>
	<code>39 \skip_new:N \l__xtemplate_tmp_skip</code>
<code>\l__xtemplate_tmp_tl</code>	A scratch variable for comparisons and so on.
	<code>40 \tl_new:N \l__xtemplate_tmp_tl</code>

12.2 Variant of prop functions

`\prop_get:NoNTF` In some cases, we need to expand the key, and get the corresponding value in a property list if it exists.

```

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 }

```

12.3 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.

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

```

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 }

```

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

```

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}
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 “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:nTF` Tests for the first token in a string being `\KeyValue`, where `\EvaluateNow` is not important.
`__xtemplate_if_key_value:oTF`

```

85 \prg_new_conditional:Npnn \__xtemplate_if_key_value:n #1 { T , F , TF }
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 }
92 \cs_generate_variant:Nn \__xtemplate_if_key_value:nF { o }
93 \cs_generate_variant:Nn \__xtemplate_if_key_value:nTF { o }

```

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

```

94 \prg_new_conditional:Npnn \__xtemplate_if_eval_now:n #1 { TF }
95 {
96     \str_if_eq:noTF { \EvaluateNow } { \tl_head:w #1 \q_nil \q_stop }
97     { \prg_return_true: }
98     { \prg_return_false: }
99 }

```

`__xtemplate_if_instance_exist:nnnTF` Testing for an instance is collection dependent.

```

100 \prg_new_conditional:Npnn \__xtemplate_if_instance_exist:nnn #1#2#3
101 { T, F, TF }
102 {
103     \cs_if_exist:cTF { \c__xtemplate_instances_root_tl #1 / #2 / #3 }
104     { \prg_return_true: }
105     { \prg_return_false: }
106 }

```

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

```

107 \prg_new_conditional:Npnn \__xtemplate_if_use_template:n #1 { TF }
108 {
109     \str_if_eq:noTF { \UseTemplate } { \tl_head:w #1 \q_nil \q_stop }
110     { \prg_return_true: }
111     { \prg_return_false: }
112 }

```

12.4 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”
`__xtemplate_store_keytypes:n` lists for the template being created.

```

\__xtemplate_store_restrictions:n
\__xtemplate_store_values:n
\__xtemplate_store_vars:n
113 \cs_new_protected:Npn \__xtemplate_store_defaults:n #1
114 {
115     \prop_gc_clear_new:c { \c__xtemplate_defaults_root_tl #1 }
116     \prop_gset_eq:cN { \c__xtemplate_defaults_root_tl #1 }
117     \l__xtemplate_values_prop

```

```

118 }
119 \cs_new_protected:Npn \__xtemplate_store_keytypes:n #1
120 {
121   \prop_gclear_new:c { \c__xtemplate_keytypes_root_tl #1 }
122   \prop_gset_eq:cN { \c__xtemplate_keytypes_root_tl #1 }
123   \l__xtemplate_keytypes_prop
124   \seq_gclear_new:c { \c__xtemplate_key_order_root_tl #1 }
125   \seq_gset_eq:cN { \c__xtemplate_key_order_root_tl #1 }
126   \l__xtemplate_key_order_seq
127 }
128 \cs_new_protected:Npn \__xtemplate_store_values:n #1
129 {
130   \prop_gclear_new:c { \c__xtemplate_values_root_tl #1 }
131   \prop_gset_eq:cN { \c__xtemplate_values_root_tl #1 }
132   \l__xtemplate_values_prop
133 }
134 \cs_new_protected:Npn \__xtemplate_store_restrictions:n #1
135 {
136   \clist_gclear_new:c { \c__xtemplate_restrict_root_tl #1 }
137   \clist_gset_eq:cN { \c__xtemplate_restrict_root_tl #1 }
138   \l__xtemplate_restrict_clist
139 }
140 \cs_new_protected:Npn \__xtemplate_store_vars:n #1
141 {
142   \prop_gclear_new:c { \c__xtemplate_vars_root_tl #1 }
143   \prop_gset_eq:cN { \c__xtemplate_vars_root_tl #1 }
144   \l__xtemplate_vars_prop
145 }

```

<pre> __xtemplate_recover_defaults:n __xtemplate_recover_keytypes:n __xtemplate_recover_restrictions:n __xtemplate_recover_values:n __xtemplate_recover_vars:n </pre>	<p>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.</p> <pre> 146 \cs_new_protected:Npn __xtemplate_recover_defaults:n #1 147 { 148 \prop_set_eq:Nc \l__xtemplate_values_prop 149 { \c__xtemplate_defaults_root_tl #1 } 150 } 151 \cs_new_protected:Npn __xtemplate_recover_keytypes:n #1 152 { 153 \prop_set_eq:Nc \l__xtemplate_keytypes_prop 154 { \c__xtemplate_keytypes_root_tl #1 } 155 \seq_set_eq:Nc \l__xtemplate_key_order_seq 156 { \c__xtemplate_key_order_root_tl #1 } 157 } 158 \cs_new_protected:Npn __xtemplate_recover_restrictions:n #1 159 { 160 \clist_set_eq:Nc \l__xtemplate_restrict_clist 161 { \c__xtemplate_restrict_root_tl #1 } 162 } 163 \cs_new_protected:Npn __xtemplate_recover_values:n #1 164 { </pre>
--	--

```

165     \prop_set_eq:Nc \l__xtemplate_values_prop
166     { \c__xtemplate_values_root_tl #1 }
167   }
168 \cs_new_protected:Npn \__xtemplate_recover_vars:n #1
169 {
170     \prop_set_eq:Nc \l__xtemplate_vars_prop
171     { \c__xtemplate_vars_root_tl #1 }
172 }

```

12.5 Creating new object types

`__xtemplate_declare_object_type:nn` 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.

```

173 \cs_new_protected:Npn \__xtemplate_declare_object_type:nn #1#2
174 {
175     \int_set:Nn \l__xtemplate_tmp_int {#2}
176     \bool_if:nTF
177     {
178         \int_compare_p:nNn {#2} > \c_nine ||
179         \int_compare_p:nNn {#2} < \c_zero
180     }
181     {
182         \msg_error:nnxx { xtemplate } { bad-number-of-arguments }
183         {#1} { \exp_not:V \l__xtemplate_tmp_int }
184     }
185     {
186         \msg_info:nnxx { xtemplate } { declare-object-type }
187         {#1} {#2}
188         \prop_gput:NnV \g__xtemplate_object_type_prop {#1}
189         \l__xtemplate_tmp_int
190     }
191 }

```

12.6 Design part of template declaration

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

`__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.

```

192 \cs_new_protected:Npn \__xtemplate_declare_template_keys:nnnn #1#2#3#4
193 {

```



```

194 \__xtemplate_execute_if_type_exist:nT {#1}
195 {
196   \__xtemplate_execute_if_arg_agree:nnT {#1} {#3}
197   {
198     \prop_clear:N \l__xtemplate_values_prop
199     \prop_clear:N \l__xtemplate_keytypes_prop
200     \seq_clear:N \l__xtemplate_key_order_seq
201     \keyval_parse:NNn
202     \__xtemplate_parse_keys_elt:n \__xtemplate_parse_keys_elt:nn {#4}
203     \__xtemplate_store_defaults:n { #1 / #2 }
204     \__xtemplate_store_keytypes:n { #1 / #2 }
205   }
206 }
207 }

```

__xtemplate_parse_keys_elt:n
 __xtemplate_parse_keys_elt_aux:n
 __xtemplate_parse_keys_elt_aux:

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

```

208 \cs_new_protected:Npn \__xtemplate_parse_keys_elt:n #1
209 {
210   \__xtemplate_split_keytype:n {#1}
211   \bool_if:NF \l__xtemplate_error_bool
212   {
213     \__xtemplate_execute_if_keytype_exist:oT \l__xtemplate_keytype_tl
214     {
215       \seq_map_function:NN \c__xtemplate_keytypes_arg_seq
216       \__xtemplate_parse_keys_elt_aux:n
217       \bool_if:NF \l__xtemplate_error_bool
218       {
219         \seq_if_in:NoTF \l__xtemplate_key_order_seq
220         \l__xtemplate_key_name_tl
221         {
222           \msg_error:nnx { xtemplate }
223           { duplicate-key-interface }
224           { \l__xtemplate_key_name_tl }
225         }
226         { \__xtemplate_parse_keys_elt_aux: }
227       }
228     }
229   }
230 }
231 \cs_new_protected_nopar:Npn \__xtemplate_parse_keys_elt_aux:n #1
232 {
233   \str_if_eq:onT \l__xtemplate_keytype_tl {#1}
234   {
235     \tl_if_empty:NT \l__xtemplate_keytype_arg_tl

```

```

236     {
237         \msg_error:nnx { xtemplate }
238         { keytype-requires-argument } {#1}
239         \bool_set_true:N \l__xtemplate_error_bool
240         \seq_map_break:
241     }
242 }
243 }
244 \cs_new_nopar:Npn \__xtemplate_parse_keys_elt_aux:
245 {
246     \tl_set:Nx \l__xtemplate_tmp_tl
247     {
248         \l__xtemplate_keytype_tl
249         \tl_if_empty:NF \l__xtemplate_keytype_arg_tl
250         { { \l__xtemplate_keytype_arg_tl } }
251     }
252     \prop_put:Noo \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
253     \l__xtemplate_tmp_tl
254     \seq_put_right:No \l__xtemplate_key_order_seq \l__xtemplate_key_name_tl
255     \str_if_eq:onT \l__xtemplate_keytype_tl { choice }
256     {
257         \clist_if_in:NnT \l__xtemplate_keytype_arg_tl { unknown }
258         { \msg_error:nn { xtemplate } { choice-unknown-reserved } }
259     }
260 }

```

__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.

```

261 \cs_new_protected:Npn \__xtemplate_parse_keys_elt:nn #1#2
262 {
263     \__xtemplate_parse_keys_elt:n {#1}
264     \use:c { __xtemplate_store_value_ \l__xtemplate_keytype_tl :n } {#2}
265 }

```

__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

```

266 \group_begin:
267 \char_set_lccode:nn { \@ } { \@: }
268 \char_set_catcode_other:N \@
269 \tl_to_lowercase:n
270 {
271     \group_end:
272     \cs_new_protected:Npn \__xtemplate_split_keytype:n #1
273     {
274         \bool_set_false:N \l__xtemplate_error_bool
275         \tl_set:Nn \l__xtemplate_tmp_tl {#1}

```

```

276 \tl_remove_all:Nn \l__xtemplate_tmp_tl { ~ }
277 \tl_replace_all:Nnn \l__xtemplate_tmp_tl { : } { @ }
278 \tl_if_in:onTF \l__xtemplate_tmp_tl { @ }
279 {
280   \tl_clear:N \l__xtemplate_key_name_tl
281   \exp_after:wN \__xtemplate_split_keytype_aux:w
282     \l__xtemplate_tmp_tl \q_stop
283 }
284 {
285   \bool_set_true:N \l__xtemplate_error_bool
286   \msg_error:nnx { xtemplate } { missing-keytype } {#1}
287 }
288 }
289 \cs_new_protected:Npn \__xtemplate_split_keytype_aux:w #1 @ #2 \q_stop
290 {
291   \tl_put_right:Nx \l__xtemplate_key_name_tl { \tl_to_str:n {#1} }
292   \tl_if_in:nnTF {#2} { @ }
293   {
294     \tl_put_right:Nn \l__xtemplate_key_name_tl { @ }
295     \__xtemplate_split_keytype_aux:w #2 \q_stop
296   }
297   {
298     \tl_if_empty:NTF \l__xtemplate_key_name_tl
299       { \msg_error:nnx { xtemplate } { empty-key-name } { @ #2 } }
300       { \__xtemplate_split_keytype_arg:n {#2} }
301   }
302 }
303 }

```

`__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).

```

304 \cs_new_protected:Npn \__xtemplate_split_keytype_arg:n #1
305 {
306   \tl_set:Nn \l__xtemplate_keytype_tl {#1}
307   \tl_clear:N \l__xtemplate_keytype_arg_tl
308   \cs_set_protected_nopar:Npn \__xtemplate_split_keytype_arg_aux:n ##1
309   {
310     \tl_if_in:nnT {#1} {##1}
311     {
312       \cs_set:Npn \__xtemplate_split_keytype_arg_aux:w
313         #####1 ##1 #####2 \q_stop
314       {
315         \tl_if_empty:nT {#####1}
316         {

```

```

317         \tl_set:Nn \l__xtemplate_keytype_tl {##1}
318         \tl_set:Nn \l__xtemplate_keytype_arg_tl {###2}
319         \seq_map_break:
320     }
321 }
322 \__xtemplate_split_keytype_arg_aux:w #1 \q_stop
323 }
324 }
325 \seq_map_function:NN \c__xtemplate_keytypes_arg_seq
326 \__xtemplate_split_keytype_arg_aux:n
327 }
328 \cs_generate_variant:Nn \__xtemplate_split_keytype_arg:n { o }
329 \cs_new_nopar:Npn \__xtemplate_split_keytype_arg_aux:n #1 { }
330 \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 ??.)

12.6.1 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`).

```

331 \cs_new_protected:Npn \__xtemplate_store_value_boolean:n #1
332 {
333     \__xtemplate_if_eval_now:nTF {#1}
334     {
335         \bool_if:cTF { c_ #1 _bool }
336         {
337             \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl
338             { true }
339         }
340         {
341             \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl
342             { false }
343         }
344     }
345     {
346         \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl {#1}
347     }
348 }

```

(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
349 \cs_new_protected:Npn \_xtemplate_store_value_code:n #1
350 { \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl {#1} }
351 \cs_new_eq:NN \_xtemplate_store_value_choice:n \_xtemplate_store_value_code:n
352 \cs_new_eq:NN \_xtemplate_store_value_commalist:n \_xtemplate_store_value_code:n
353 \cs_new_eq:NN \_xtemplate_store_value_function:n \_xtemplate_store_value_code:n
354 \cs_new_eq:NN \_xtemplate_store_value_instance:n \_xtemplate_store_value_code:n
355 \cs_new_eq:NN \_xtemplate_store_value_real:n \_xtemplate_store_value_code:n
356 \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 ??.)

```

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

```

\_xtemplate_store_value_integer:n
\_xtemplate_store_value_length:n
\_xtemplate_store_value_muskip:n
\_xtemplate_store_value_skip:n
357 \cs_new_protected:Npn \_xtemplate_store_value_integer:n #1
358 {
359   \_xtemplate_if_eval_now:nTF {#1}
360   {
361     \int_set:Nn \l__xtemplate_tmp_int {#1}
362     \prop_put:NVV \l__xtemplate_values_prop \l__xtemplate_key_name_int
363       \l__xtemplate_tmp_int
364   }
365   {
366     \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl {#1}
367   }
368 }
369 \cs_new_protected:Npn \_xtemplate_store_value_length:n #1
370 {
371   \_xtemplate_if_eval_now:nTF {#1}
372   {
373     \dim_set:Nn \l__xtemplate_tmp_dim {#1}
374     \prop_put:NVV \l__xtemplate_values_prop \l__xtemplate_key_name_tl
375       \l__xtemplate_tmp_dim
376   }
377   {
378     \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl {#1}
379   }
380 }
381 \cs_new_protected:Npn \_xtemplate_store_value_muskip:n #1
382 {
383   \_xtemplate_if_eval_now:nTF {#1}
384   {
385     \muskip_set:Nn \l__xtemplate_tmp_muskip {#1}
386     \prop_put:NVV \l__xtemplate_values_prop \l__xtemplate_key_name_tl
387       \l__xtemplate_tmp_muskip
388   }

```

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”.

```

389     {
390         \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl {#1}
391     }
392 }
393 \cs_new_protected:Npn \__xtemplate_store_value_skip:n #1
394 {
395     \__xtemplate_if_eval_now:nTF {#1}
396     {
397         \skip_set:Nn \l__xtemplate_tmp_skip {#1}
398         \prop_put:NVV \l__xtemplate_values_prop \l__xtemplate_key_name_tl
399             \l__xtemplate_tmp_skip
400     }
401     {
402         \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl {#1}
403     }
404 }

```

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

12.7 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.

```

405 \cs_new_protected:Npn \__xtemplate_declare_template_code:nnnnn #1#2#3#4#5
406 {
407     \__xtemplate_execute_if_type_exist:nT {#1}
408     {
409         \__xtemplate_execute_if_arg_agree:nnT {#1}{#3}
410         {
411             \__xtemplate_if_keys_exist:nnT {#1} {#2}
412             {
413                 \__xtemplate_store_key_implementation:nnn {#1} {#2} {#4}
414                 \cs_generate_from_arg_count:cNnn
415                     { \c__xtemplate_code_root_tl #1 / #2 }
416                 \cs_gset_protected:Npn {#3} {#5}
417             }
418         }
419     }
420 }

```

(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.

```

421 \cs_new_protected:Npn \__xtemplate_store_key_implementation:nnn #1#2#3

```

```

422 {
423     \__xtemplate_recover_defaults:n { #1 / #2 }
424     \__xtemplate_recover_keytypes:n { #1 / #2 }
425     \prop_clear:N \l__xtemplate_vars_prop
426     \keyval_parse:NNn
427     \__xtemplate_parse_vars_elt:n \__xtemplate_parse_vars_elt:nn {#3}
428     \__xtemplate_store_vars:n { #1 / #2 }
429     \clist_clear:N \l__xtemplate_restrict_clist
430     \__xtemplate_store_restrictions:n { #1 / #2 }
431     \prop_map_inline:Nn \l__xtemplate_keytypes_prop
432     {
433         \msg_error:nnxxx { xtemplate } { key-not-implemented }
434         {##1} {#2} {#1}
435     }
436 }

```

(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.

```

437 \cs_new_protected:Npn \__xtemplate_parse_vars_elt:n #1
438 { \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.

```

439 \cs_new_protected:Npn \__xtemplate_parse_vars_elt:nn #1#2
440 {
441     \tl_set:Nx \l__xtemplate_key_name_tl { \tl_to_str:n {#1} }
442     \tl_remove_all:Nn \l__xtemplate_key_name_tl { ~ }
443     \prop_get:NoNTF
444     \l__xtemplate_keytypes_prop
445     \l__xtemplate_key_name_tl
446     \l__xtemplate_keytype_tl
447     {
448         \__xtemplate_split_keytype_arg:o \l__xtemplate_keytype_tl
449         \__xtemplate_parse_vars_elt_aux:n {#2}
450         \prop_remove:NV \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
451     }
452     { \msg_error:nnx { xtemplate } { unknown-key } {#1} }
453 }

```

(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 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.

__xtemplate_parse_vars_elt_aux:w

```

454 \cs_new_protected:Npn \__xtemplate_parse_vars_elt_aux:n #1
455 {

```

```

456 \str_if_eq:onTF \l__xtemplate_keytype_tl { choice }
457 { \__xtemplate_implement_choices:n {#1} }
458 {
459   \str_if_eq:onTF \l__xtemplate_keytype_tl { code }
460   {
461     \prop_put:Non \l__xtemplate_vars_prop
462     \l__xtemplate_key_name_tl {#1}
463   }
464   {
465     \tl_if_single:nTF {#1}
466     {
467       \cs_if_exist:NF #1
468       { \__xtemplate_create_variable:N #1 }
469       \prop_put:Non \l__xtemplate_vars_prop
470       \l__xtemplate_key_name_tl {#1}
471     }
472     {
473       \tl_if_in:nnTF {#1} { global }
474       { \__xtemplate_parse_vars_elt_aux:w #1 \q_stop }
475       {
476         \msg_error:nnx { xtemplate } { bad-variable }
477         { \tl_to_str:n {#1} }
478       }
479     }
480   }
481 }
482 }
483 \cs_new_protected:Npn \__xtemplate_parse_vars_elt_aux:w #1 global #2 \q_stop
484 {
485   \tl_if_empty:nTF {#1}
486   {
487     \tl_if_single:nTF {#2}
488     {
489       \cs_if_exist:NF #2
490       { \__xtemplate_create_variable:N #2 }
491       \prop_put:Non \l__xtemplate_vars_prop
492       \l__xtemplate_key_name_tl { #1 global #2 }
493     }
494     {
495       \msg_error:nnx { xtemplate } { bad-variable }
496       { \tl_to_str:n { #1 global #2 } }
497     }
498   }
499   {
500     \msg_error:nnx { xtemplate } { bad-variable }
501     { \tl_to_str:n { #1 global #2 } }
502   }
503 }

```

(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.

```

504 \cs_new_protected_nopar:Npn \_xtemplate_create_variable:N #1
505 {
506   \str_case:onnn \l__xtemplate_keytype_tl
507   {
508     { boolean } { \bool_new:N #1 }
509     { commalist } { \clist_new:N #1 }
510     { function } { \cs_new:Npn #1 { } }
511     { instance } { \cs_new_protected:Npn #1 { } }
512     { integer } { \int_new:N #1 }
513     { length } { \dim_new:N #1 }
514     { real } { \fp_new:N #1 }
515     { tokenlist } { \tl_new:N #1 }
516   }
517   { \use:c { \l__xtemplate_keytype_tl _ new:N } #1 }
518 }

```

(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
`_xtemplate_implement_choices_default:` standard parser is called.

```

519 \cs_new_protected:Npn \_xtemplate_implement_choices:n #1
520 {
521   \clist_set_eq:NN \l__xtemplate_tmp_clist \l__xtemplate_keytype_arg_tl
522   \prop_put:Non \l__xtemplate_vars_prop \l__xtemplate_key_name_tl { }
523   \keyval_parse:NNn
524     \_xtemplate_implement_choice_elt:n \_xtemplate_implement_choice_elt:nn
525     {#1}
526   \prop_get:NoNT \l__xtemplate_values_prop \l__xtemplate_key_name_tl
527     \l__xtemplate_tmp_tl
528   { \_xtemplate_implement_choices_default: }
529   \clist_if_empty:NF \l__xtemplate_tmp_clist
530   {
531     \clist_map_inline:Nn \l__xtemplate_tmp_clist
532     {
533       \msg_error:nnx { xtemplate } { choice-not-implemented }
534       {#1}
535     }
536   }
537 }

```

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

```

538 \cs_new_protected_nopar:Npn \_xtemplate_implement_choices_default:
539 {
540   \tl_set:Nx \l__xtemplate_tmp_tl
541     { \l__xtemplate_key_name_tl \c_space_tl \l__xtemplate_tmp_tl }
542   \prop_if_in:NoF \l__xtemplate_vars_prop \l__xtemplate_tmp_tl
543   {

```

```

544     \tl_set:Nx \l__xtemplate_tmp_tl
545     { \l__xtemplate_key_name_tl \c_space_tl \l__xtemplate_tmp_tl }
546     \prop_if_in:NoF \l__xtemplate_vars_prop \l__xtemplate_tmp_tl
547     {
548         \prop_get:NoN \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
549         \l__xtemplate_tmp_tl
550         \__xtemplate_split_keytype_arg:o \l__xtemplate_tmp_tl
551         \prop_get:NoN \l__xtemplate_values_prop \l__xtemplate_key_name_tl
552         \l__xtemplate_tmp_tl
553         \msg_error:nnxxx { xtemplate } { unknown-default-choice }
554         { \l__xtemplate_key_name_tl } { \l__xtemplate_key_name_tl }
555         { \l__xtemplate_keytype_arg_tl }
556     }
557 }
558 }

```

(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 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.

```

559 \cs_new_protected:Npn \__xtemplate_implement_choice_elt:n #1
560 {
561     \clist_if_empty:NTF \l__xtemplate_tmp_clist
562     {
563         \str_if_eq:nnF {#1} { unknown }
564         {
565             \prop_get:NoN \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
566             \l__xtemplate_tmp_tl
567             \__xtemplate_split_keytype_arg:o \l__xtemplate_tmp_tl
568             \msg_error:nnxxx { xtemplate } { unknown-choice }
569             { \l__xtemplate_key_name_tl } {#1}
570             { \l__xtemplate_keytype_arg_tl }
571         }
572     }
573     {
574         \clist_if_in:NnTF \l__xtemplate_tmp_clist {#1}
575         { \clist_remove_all:Nn \l__xtemplate_tmp_clist {#1} }
576         {
577             \prop_get:NoN \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
578             \l__xtemplate_tmp_tl
579             \__xtemplate_split_keytype_arg:o \l__xtemplate_tmp_tl
580             \msg_error:nnxxx { xtemplate } { unknown-choice }
581             { \l__xtemplate_key_name_tl } {#1}
582             { \l__xtemplate_keytype_arg_tl }
583         }
584     }
585 }
586 \cs_new_protected:Npn \__xtemplate_implement_choice_elt:nn #1#2

```

```

587 {
588   \__xtemplate_implement_choice_elt:n {#1}
589   \tl_set:Nx \l__xtemplate_tmp_tl
590     { \l__xtemplate_key_name_tl \c_space_tl #1 }
591   \prop_put:Non \l__xtemplate_vars_prop \l__xtemplate_tmp_tl {#2}
592 }

```

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

12.8 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.

```

593 \cs_new_protected:Npn \__xtemplate_declare_restricted:nnnn #1#2#3#4
594 {
595   \__xtemplate_if_keys_exist:nnT {#1} {#2}
596   {
597     \__xtemplate_set_template_eq:nn { #1 / #3 } { #1 / #2 }
598     \bool_set_true:N \l__xtemplate_restrict_bool
599     \__xtemplate_edit_defaults_aux:nnn {#1} {#3} {#4}
600   }
601 }

```

(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 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.

__xtemplate_edit_defaults_aux:nnn

```

602 \cs_new_protected:Npn \__xtemplate_edit_defaults:nnn
603 {
604   \bool_set_false:N \l__xtemplate_restrict_bool
605   \__xtemplate_edit_defaults_aux:nnn
606 }
607 \cs_new_protected:Npn \__xtemplate_edit_defaults_aux:nnn #1#2#3
608 {
609   \__xtemplate_if_keys_exist:nnT {#1} {#2}
610   {
611     \__xtemplate_recover_defaults:n { #1 / #2 }
612     \__xtemplate_recover_restrictions:n { #1 / #2 }
613     \__xtemplate_parse_values:nn { #1 / #2 } {#3}
614     \__xtemplate_store_defaults:n { #1 / #2 }
615     \__xtemplate_store_restrictions:n { #1 / #2 }
616   }
617 }

```

(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.

```

618 \cs_new_protected:Npn \__xtemplate_parse_values:nn #1#2
619 {
620   \__xtemplate_recover_keytypes:n {#1}
621   \clist_clear:N \l__xtemplate_restrict_clist
622   \keyval_parse:NNn
623     \__xtemplate_parse_values_elt:n \__xtemplate_parse_values_elt:nn {#2}
624 }

```

(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.

```

625 \cs_new_protected:Npn \__xtemplate_parse_values_elt:n #1
626 {
627   \bool_set_true:N \l__xtemplate_error_bool
628   \msg_error:nnx { xtemplate } { key-no-value } {#1}
629 }

```

(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.

`__xtemplate_parse_values_elt_aux:n`

```

630 \cs_new_protected:Npn \__xtemplate_parse_values_elt:nn #1#2
631 {
632   \tl_set:Nx \l__xtemplate_key_name_tl { \tl_to_str:n {#1} }
633   \tl_remove_all:Nn \l__xtemplate_key_name_tl { ~ }
634   \prop_get:NoNTF \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
635     \l__xtemplate_tmp_tl
636   {
637     \bool_if:NTF \l__xtemplate_restrict_bool
638       {
639         \clist_if_in:NoF \l__xtemplate_restrict_clist
640           \l__xtemplate_key_name_tl
641           { \__xtemplate_parse_values_elt_aux:n {#2} }
642       }
643     { \__xtemplate_parse_values_elt_aux:n {#2} }
644   }
645   {
646     \msg_error:nnx { xtemplate } { unknown-key }
647     { \l__xtemplate_key_name_tl }
648   }
649 }
650 \cs_new_protected:Npn \__xtemplate_parse_values_elt_aux:n #1
651 {
652   \clist_put_right:No \l__xtemplate_restrict_clist \l__xtemplate_key_name_tl
653   \__xtemplate_split_keytype_arg:o \l__xtemplate_tmp_tl
654   \use:c { __xtemplate_store_value_ \l__xtemplate_keytype_tl :n } {#1}
655 }

```

(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.

```

656 \cs_new_protected:Npn \_xtemplate_set_template_eq:nn #1#2
657 {
658   \_xtemplate_recover_defaults:n {#2}
659   \_xtemplate_store_defaults:n {#1}
660   \_xtemplate_recover_keytypes:n {#2}
661   \_xtemplate_store_keytypes:n {#1}
662   \_xtemplate_recover_vars:n {#2}
663   \_xtemplate_store_vars:n {#1}
664   \cs_gset_eq:cc { \c__xtemplate_code_root_tl #1 }
665     { \c__xtemplate_code_root_tl #2 }
666 }

```

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

12.9 Creating instances of templates

`_xtemplate_declare_instance:nnnnn` 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.

`_xtemplate_declare_instance_aux:nnnnn`

```

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

```

```

692     {
693         \cs_set_eq:cc
694         { \c__xtemplate_instances_root_tl #1 /      / #4 }
695         { \c__xtemplate_instances_root_tl #1 / #3 / #4 }
696     }
697 }
698 }

```

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

`__xtemplate_edit_instance:nnnn`
`__xtemplate_edit_instance_aux:nnnnn`
`__xtemplate_edit_instance_aux:nonnn`

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.

```

699 \cs_new_protected:Npn \__xtemplate_edit_instance:nnnn #1#2#3
700 {
701     \__xtemplate_if_instance_exist:nnnTF {#1} {#2} {#3}
702     {
703         \__xtemplate_recover_values:n { #1 / #2 / #3 }
704         \prop_get:NnN \l__xtemplate_values_prop { from-template }
705         \l__xtemplate_tmp_tl
706         \__xtemplate_edit_instance_aux:nonnn {#1} \l__xtemplate_tmp_tl
707         {#2} {#3}
708     }
709     {
710         \msg_error:nnxx { xtemplate } { unknown-instance }
711         {#1} {#3}
712     }
713 }
714 \cs_new_protected:Npn \__xtemplate_edit_instance_aux:nnnnn #1#2
715 {
716     \__xtemplate_recover_vars:n { #1 / #2 }
717     \__xtemplate_declare_instance_aux:nnnnn {#1} {#2}
718 }
719 \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”.

```

720 \cs_new_protected_nopar:Npn \__xtemplate_convert_to_assignments:
721 {
722     \tl_clear:N \l__xtemplate_assignments_tl
723     \seq_map_function:NN \l__xtemplate_key_order_seq
724     \__xtemplate_convert_to_assignments_aux:n
725 }
726 \cs_new_protected:Npn \__xtemplate_convert_to_assignments_aux:n #1
727 {
728     \prop_get:NnN \l__xtemplate_keytypes_prop {#1} \l__xtemplate_tmp_tl
729     \__xtemplate_convert_to_assignments_aux:no {#1} \l__xtemplate_tmp_tl
730 }

```

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.

```

731 \cs_new_protected:Npn \__xtemplate_convert_to_assignments_aux:nn #1#2
732 {
733   \prop_get:NnNT \l__xtemplate_values_prop {#1} \l__xtemplate_value_tl
734   {
735     \prop_get:NnNTF \l__xtemplate_vars_prop {#1} \l__xtemplate_var_tl
736     {
737       \__xtemplate_split_keytype_arg:n {#2}
738       \str_if_eq:onF \l__xtemplate_keytype_tl { choice }
739       {
740         \str_if_eq:onF \l__xtemplate_keytype_tl { code }
741         { \__xtemplate_find_global: }
742       }
743       \tl_set:Nn \l__xtemplate_key_name_tl {#1}
744       \use:c { __xtemplate_assign_ \l__xtemplate_keytype_tl : }
745     }
746     { \msg_error:nxx { xtemplate } { unknown-attribute } {#1} }
747   }
748 }
749 \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.

```

\__xtemplate_find_global_aux:w
750 \cs_new_protected_nopar:Npn \__xtemplate_find_global:
751 {
752   \bool_set_false:N \l__xtemplate_global_bool
753   \tl_if_in:onT \l__xtemplate_var_tl { global }
754   {
755     \exp_after:wN \__xtemplate_find_global_aux:w \l__xtemplate_var_tl \q_stop
756   }
757 }
758 \cs_new_protected:Npn \__xtemplate_find_global_aux:w #1 global #2 \q_stop
759 {
760   \tl_set:Nn \l__xtemplate_var_tl {#2}
761   \bool_set_true:N \l__xtemplate_global_bool
762 }

```

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

12.10 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.

```

763 \cs_new_protected:Npn \__xtemplate_use_template:nnn #1#2#3
764 {
765   \__xtemplate_execute_if_code_exist:nnT {#1} {#2}
766   {
767     \__xtemplate_recover_defaults:n { #1 / #2 }
768     \__xtemplate_recover_vars:n { #1 / #2 }
769     \__xtemplate_parse_values:nn { #1 / #2 } {#3}
770     \__xtemplate_convert_to_assignments:
771     \use:c { \c__xtemplate_code_root_tl #1 / #2 }
772   }
773 }

```

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

12.11 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. If there is, then we need to allow for the fact that the recovered value here will *not* be expandable, so needs to be converted to something that is.

```

774 \cs_new_protected_nopar:Npn \__xtemplate_assign_boolean:
775 {
776   \bool_if:NTF \l__xtemplate_global_bool
777   { \__xtemplate_assign_boolean_aux:n { bool_gset } }
778   { \__xtemplate_assign_boolean_aux:n { bool_set } }
779 }
780 \cs_new_protected_nopar:Npn \__xtemplate_assign_boolean_aux:n #1
781 {
782   \__xtemplate_if_key_value:oTF \l__xtemplate_value_tl
783   {
784     \__xtemplate_key_to_value:
785     \tl_put_right:Nx \l__xtemplate_assignments_tl
786     {
787       \exp_not:c { #1 _eq:NN }
788       \exp_not:o \l__xtemplate_var_tl
789       \exp_not:o \l__xtemplate_value_tl
790     }
791   }
792   {
793     \tl_put_right:Nx \l__xtemplate_assignments_tl
794     {
795       \exp_not:c { #1 _ \l__xtemplate_value_tl :N }
796       \exp_not:o \l__xtemplate_var_tl
797     }
798   }
799 }

```

(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.

```

\__xtemplate_assign_choice_aux:n
\__xtemplate_assign_choice_aux:o
800 \cs_new_protected_nopar:Npn \__xtemplate_assign_choice:
801 {
802   \__xtemplate_assign_choice_aux:xF
803   { \l__xtemplate_key_name_tl \c_space_tl \l__xtemplate_value_tl }
804   {
805     \__xtemplate_assign_choice_aux:xF
806     { \l__xtemplate_key_name_tl \c_space_tl unknown }
807     {
808       \prop_get:NoN \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
809       \l__xtemplate_tmp_tl
810       \__xtemplate_split_keytype_arg:o \l__xtemplate_tmp_tl
811       \msg_error:nnxxx { xtemplate } { unknown-choice }
812       { \l__xtemplate_key_name_tl } { \l__xtemplate_value_tl }
813       { \l__xtemplate_keytype_arg_tl }
814     }
815   }
816 }
817 \cs_new_protected_nopar:Npn \__xtemplate_assign_choice_aux:nF #1
818 {
819   \prop_get:NnNTF
820   \l__xtemplate_vars_prop
821   {#1}
822   \l__xtemplate_tmp_tl
823   { \tl_put_right:No \l__xtemplate_assignments_tl \l__xtemplate_tmp_tl }
824 }
825 \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 `\AssignTemplateKeys` is called. So the appropriate definition then use is created in the token list variable.

```

826 \cs_new_protected_nopar:Npn \__xtemplate_assign_code:
827 {
828   \tl_put_right:Nx \l__xtemplate_assignments_tl
829   {
830     \cs_set_protected:Npn \__xtemplate_assign_code:n \exp_not:n {##1}
831     { \exp_not:o \l__xtemplate_var_tl }
832     \__xtemplate_assign_code:n { \exp_not:o \l__xtemplate_value_tl }
833   }
834 }
835 \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
836 \cs_new_protected_nopar:Npn \_xtemplate_assign_function:
837 {
838   \bool_if:NTF \l__xtemplate_global_bool

```

```

839     { \__xtemplate_assign_function_aux:N \cs_gset:Npn }
840     { \__xtemplate_assign_function_aux:N \cs_set:Npn }
841   }
842 \cs_new_protected_nopar:Npn \__xtemplate_assign_function_aux:N #1
843 {
844   \tl_put_right:Nx \l__xtemplate_assignments_tl
845   {
846     \cs_generate_from_arg_count:NNnn
847     \exp_not:o \l__xtemplate_var_tl
848     \exp_not:N #1
849     { \exp_not:o \l__xtemplate_keytype_arg_tl }
850     { \exp_not:o \l__xtemplate_value_tl }
851   }
852 }

```

(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 are made at this stage, so if the instance is not valid then errors will arise later.

__xtemplate_assign_instance_aux:N

```

853 \cs_new_protected_nopar:Npn \__xtemplate_assign_instance:
854 {
855   \bool_if:NTF \l__xtemplate_global_bool
856   { \__xtemplate_assign_instance_aux:N \cs_gset_protected:Npn }
857   { \__xtemplate_assign_instance_aux:N \cs_set_protected:Npn }
858 }
859 \cs_new_protected_nopar:Npn \__xtemplate_assign_instance_aux:N #1
860 {
861   \tl_put_right:Nx \l__xtemplate_assignments_tl
862   {
863     \exp_not:N #1 \exp_not:o \l__xtemplate_var_tl
864     {
865       \__xtemplate_use_instance:nn
866       { \exp_not:o \l__xtemplate_keytype_arg_tl }
867       { \exp_not:o \l__xtemplate_value_tl }
868     }
869   }
870 }

```

(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:

```

871 \cs_new_protected_nopar:Npn \__xtemplate_assign_integer:
872 {
873   \bool_if:NTF \l__xtemplate_global_bool
874   { \__xtemplate_assign_variable:N \int_gset:Nn }
875   { \__xtemplate_assign_variable:N \int_set:Nn }
876 }
877 \cs_new_protected_nopar:Npn \__xtemplate_assign_length:
878 {
879   \bool_if:NTF \l__xtemplate_global_bool

```

```

880     { \__xtemplate_assign_variable:N \dim_gset:Nn }
881     { \__xtemplate_assign_variable:N \dim_set:Nn }
882 }
883 \cs_new_protected_nopar:Npn \__xtemplate_assign_muskip:
884 {
885     \bool_if:NTF \l__xtemplate_global_bool
886     { \__xtemplate_assign_variable:N \muskip_gset:Nn }
887     { \__xtemplate_assign_variable:N \muskip_set:Nn }
888 }
889 \cs_new_protected_nopar:Npn \__xtemplate_assign_real:
890 {
891     \bool_if:NTF \l__xtemplate_global_bool
892     { \__xtemplate_assign_variable:N \fp_gset:Nn }
893     { \__xtemplate_assign_variable:N \fp_set:Nn }
894 }
895 \cs_new_protected_nopar:Npn \__xtemplate_assign_skip:
896 {
897     \bool_if:NTF \l__xtemplate_global_bool
898     { \__xtemplate_assign_variable:N \skip_gset:Nn }
899     { \__xtemplate_assign_variable:N \skip_set:Nn }
900 }

```

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

__xtemplate_assign_tokenlist: Life would be easy here if it were not for \KeyValue. To deal correctly with that, we
__xtemplate_assign_tokenlist_aux:NN need to allow for the recovery a stored value at point of use.

```

901 \cs_new_protected_nopar:Npn \__xtemplate_assign_tokenlist:
902 {
903     \bool_if:NTF \l__xtemplate_global_bool
904     { \__xtemplate_assign_tokenlist_aux:NN \tl_gset:NV \tl_gset:Nn }
905     { \__xtemplate_assign_tokenlist_aux:NN \tl_set:NV \tl_set:Nn }
906 }
907 \cs_new_protected_nopar:Npn \__xtemplate_assign_tokenlist_aux:NN #1#2
908 {
909     \__xtemplate_if_key_value:oTF \l__xtemplate_value_tl
910     {
911         \__xtemplate_key_to_value:
912         \tl_put_right:Nx \l__xtemplate_assignments_tl
913         {
914             #1 \exp_not:o \l__xtemplate_var_tl
915             \exp_not:o \l__xtemplate_value_tl
916         }
917     }
918     {
919         \tl_put_right:Nx \l__xtemplate_assignments_tl
920         {
921             #2 \exp_not:o \l__xtemplate_var_tl
922             { \exp_not:o \l__xtemplate_value_tl }
923         }
924     }
925 }

```

(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.

```

926 \cs_new_protected_nopar:Npn \__xtemplate_assign_commalist:
927 {
928   \bool_if:NTF \l__xtemplate_global_bool
929     { \__xtemplate_assign_tokenlist_aux:NN \clist_gset:NV \clist_gset:Nn }
930     { \__xtemplate_assign_tokenlist_aux:NN \clist_set:NV \clist_set:Nn }
931 }

```

(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.

```

932 \cs_new_protected_nopar:Npn \__xtemplate_assign_variable:N #1
933 {
934   \__xtemplate_if_key_value:oT \l__xtemplate_value_tl
935   { \__xtemplate_key_to_value: }
936   \tl_put_right:Nx \l__xtemplate_assignments_tl
937   {
938     #1 \exp_not:o \l__xtemplate_var_tl
939     { \exp_not:o \l__xtemplate_value_tl }
940   }
941 }

```

(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. There is also a need to check in case the copied value happens to be `global`.

`__xtemplate_key_to_value_auxi:w`
`__xtemplate_key_to_value_auxii:w`

```

942 \cs_new_protected_nopar:Npn \__xtemplate_key_to_value:
943 { \exp_after:wN \__xtemplate_key_to_value_auxi:w \l__xtemplate_value_tl }
944 \cs_new_protected:Npn \__xtemplate_key_to_value_auxi:w \KeyValue #1
945 {
946   \tl_set:Nx \l__xtemplate_tmp_tl { \tl_to_str:n {#1} }
947   \tl_remove_all:Nn \l__xtemplate_key_name_tl { ~ }
948   \prop_get:NoNTF
949   \l__xtemplate_vars_prop
950   \l__xtemplate_tmp_tl
951   \l__xtemplate_value_tl
952   {
953     \exp_after:wN \__xtemplate_key_to_value_auxii:w \l__xtemplate_value_tl
954     \q_mark global \q_nil \q_stop
955   }
956   {
957     \msg_error:nnx { xtemplate } { unknown-attribute }
958     { \l__xtemplate_tmp_tl }
959   }
960 }

```

```

961 \cs_new_protected:Npn \__xtemplate_key_to_value_auxii:w #1 global #2#3 \q_stop
962 {
963   \quark_if_nil:NF #2
964   { \tl_set:Nn \l__xtemplate_value_tl {#2} }
965 }

```

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

12.12 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.

```

966 \cs_new_protected:Npn \__xtemplate_use_instance:nn #1#2
967 {
968   \__xtemplate_if_use_template:nTF {#2}
969   { \__xtemplate_use_instance_aux:nNnnn {#1} #2 }
970   { \__xtemplate_use_instance_aux:nn {#1} {#2} }
971 }
972 \cs_new_protected:Npn \__xtemplate_use_instance_aux:nNnnn #1#2#3#4#5
973 {
974   \str_if_eq:nnTF {#1} {#3}
975   { \__xtemplate_use_template:nnn {#3} {#4} {#5} }
976   { \msg_error:nnxx { xtemplate } { type-mismatch } {#1} {#3} }
977 }
978 \cs_new_protected:Npn \__xtemplate_use_instance_aux:nn #1#2
979 {
980   \__xtemplate_get_collection:n {#1}
981   \__xtemplate_if_instance_exist:nnnTF
982   {#1} { \l__xtemplate_collection_tl } {#2}
983   {
984     \use:c
985     {
986       \c__xtemplate_instances_root_tl #1 /
987       \l__xtemplate_collection_tl / #2
988     }
989   }
990   {
991     \__xtemplate_if_instance_exist:nnnTF {#1} { } {#2}
992     { \use:c { \c__xtemplate_instances_root_tl #1 / / #2 } }
993     {
994       \msg_error:nnxx { xtemplate } { unknown-instance }
995       {#1} {#2}
996     }
997   }
998 }

```

(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.

```

999 \cs_new_protected:Npn \_xtemplate_use_collection:nn #1#2
1000 { \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.

```

1001 \cs_new_protected:Npn \_xtemplate_get_collection:n #1
1002 {
1003   \prop_get:NnNF \l__xtemplate_collections_prop {#1}
1004   \l__xtemplate_collection_tl
1005   { \tl_clear:N \l__xtemplate_collection_tl }
1006 }
(End definition for \_xtemplate_get_collection:n. This function is documented on page ??.)

```

12.13 Assignment manipulation

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

`_xtemplate_assignments_pop:` To actually use the assignments.

```

1007 \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.

```

1008 \cs_new_protected:Npn \_xtemplate_assignments_push:n #1
1009 { \tl_set:Nn \l__xtemplate_assignments_tl {#1} }
(End definition for \_xtemplate_assignments_push:n. This function is documented on page ??.)

```

12.14 Showing templates and instances

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

```

1010 \cs_new_protected_nopar:Npn \_xtemplate_show_code:nn #1#2
1011 { \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
1012 \cs_new_protected_nopar:Npn \_xtemplate_show_defaults:nn #1#2
1013 {
1014   \_xtemplate_if_keys_exist:nnT {#1} {#2}
1015   {
1016     \_xtemplate_recover_defaults:n { #1 / #2 }
1017     \_xtemplate_show:Nnnn \l__xtemplate_values_prop
1018     {#1} {#2} { default~values }
1019   }
1020 }
1021 \cs_new_protected_nopar:Npn \_xtemplate_show_keytypes:nn #1#2
1022 {

```

```

1023 \__xtemplate_if_keys_exist:nnT {#1} {#2}
1024 {
1025   \__xtemplate_recover_keytypes:n { #1 / #2 }
1026   \__xtemplate_show:Nnnn \l__xtemplate_keytypes_prop
1027   {#1} {#2} { interface }
1028 }
1029 }
1030 \cs_new_protected_nopar:Npn \__xtemplate_show_vars:nn #1#2
1031 {
1032   \__xtemplate_execute_if_code_exist:nnT {#1} {#2}
1033   {
1034     \__xtemplate_recover_vars:n { #1 / #2 }
1035     \__xtemplate_show:Nnnn \l__xtemplate_vars_prop
1036     {#1} {#2} { variable-mapping }
1037   }
1038 }
1039 \cs_new_protected_nopar:Npn \__xtemplate_show:Nnnn #1#2#3#4
1040 {
1041   \__msg_term:nnnnn { xtemplate }
1042   { \prop_if_empty:NTF #1 { show-no-attribute } { show-attribute } }
1043   {#2} {#3} {#4}
1044   \__msg_show_variable:n
1045   { \prop_map_function:NN #1 \__msg_show_item_unbraced:nn }
1046 }

```

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

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

```

1047 \cs_new_protected_nopar:Npn \__xtemplate_show_values:nnn #1#2#3
1048 {
1049   \__xtemplate_if_instance_exist:nnnT {#1} {#2} {#3}
1050   {
1051     \__xtemplate_recover_values:n { #1 / #2 / #3 }
1052     \prop_if_empty:NTF \l__xtemplate_values_prop
1053     {
1054       \__msg_term:nnnnn { xtemplate } { show-no-values }
1055       {#1} {#2} {#3}
1056       \__msg_show_variable:n { }
1057     }
1058     {
1059       \prop_pop:NnN \l__xtemplate_values_prop { from-template }
1060       \l__xtemplate_tmp_tl
1061       \__msg_term:nnnnnV { xtemplate } { show-values }
1062       {#1} {#2} {#3} \l__xtemplate_tmp_tl
1063       \__msg_show_variable:n
1064       {
1065         \prop_map_function:NN \l__xtemplate_values_prop
1066         \__msg_show_item_unbraced:nn
1067       }
1068     }
1069   }
1070 }

```

```

1068     }
1069   }
1070 }

```

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

12.15 Messages

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

```

1071 \msg_new:nnnn { xtemplate } { argument-number-mismatch }
1072 { Object~type~'#1'~takes~#2~argument(s). }
1073 {
1074   \c_msg_coding_error_text_tl
1075   Objects~of~type~'#1'~require~#2~argument(s).\
1076   You~have~tried~to~make~a~template~for~'#1'~
1077   with~#3~argument(s),~which~is~not~possible:~
1078   the~number~of~arguments~must~agree.
1079 }
1080 \msg_new:nnnn { xtemplate } { bad-number-of-arguments }
1081 { Bad~number~of~arguments~for~object~type~'#1'. }
1082 {
1083   \c_msg_coding_error_text_tl
1084   An~object~may~accept~between~0~and~9~arguments.\
1085   You~asked~to~use~#2~arguments:~this~is~not~supported.
1086 }
1087 \msg_new:nnnn { xtemplate } { bad-variable }
1088 { Incorrect~variable~description~'#1'. }
1089 {
1090   The~argument~'#1'~is~not~of~the~form \
1091   ~<variable>\
1092   ~or~\
1093   ~'global~<variable>'. \
1094   It~must~be~given~in~one~of~these~formats~to~be~used~in~a~template.
1095 }
1096 \msg_new:nnnn { xtemplate } { choice-not-implemented }
1097 { The~choice~'#1'~has~no~implementation. }
1098 {
1099   Each~choice~listed~in~the~interface~for~a~template~must~
1100   have~an~implementation.
1101 }
1102 \msg_new:nnnn { xtemplate } { choice-no-code }
1103 { The~choice~'#1'~requires~implementation~details. }
1104 {
1105   \c_msg_coding_error_text_tl
1106   When~creating~template~code~using~\DeclareTemplateCode,~
1107   each~choice~name~must~have~an~associated~implementation.\
1108   This~should~be~given~after~a~'= '~sign:~LaTeX~did~not~find~one.
1109 }
1110 \msg_new:nnnn { xtemplate } { duplicate-key-interface }
1111 { Key~'#1'~appears~twice~in~interface~definition~\msg_line_context:. }

```



```

1112 {
1113   \c_msg_coding_error_text_tl
1114   Each~key~can~only~have~one~interface~declared~in~a~template.\\
1115   LaTeX~found~two~interfaces~for~'#1'.
1116 }
1117 \msg_new:nnnn { xtemplate } { keytype-requires-argument }
1118 { The~key~'#1'~requires~an~argument~\msg_line_context:. }
1119 {
1120   You~should~have~put:\\
1121   \ \ <key-name>~:~'#1'~{~<argument>~} \ \
1122   but~LaTeX~did~not~find~an~<argument>.
1123 }
1124 \msg_new:nnnn { xtemplate } { invalid-keytype }
1125 { The~key~'#1'~is~missing~a~key~type~\msg_line_context:. }
1126 {
1127   \c_msg_coding_error_text_tl
1128   Each~key~in~a~template~requires~a~key~type,~given~in~the~form:\\
1129   \ \ <key>~:~<key-type>\\
1130   LaTeX~could~not~find~a~<key-type>~in~your~input.
1131 }
1132 \msg_new:nnnn { xtemplate } { key-no-value }
1133 { The~key~'#1'~has~no~value~\msg_line_context:. }
1134 {
1135   \c_msg_coding_error_text_tl
1136   When~creating~an~instance~of~a~template~
1137   every~key~listed~must~include~a~value:\\
1138   \ \ <key>~::~~<value>
1139 }
1140 \msg_new:nnnn { xtemplate } { key-no-variable }
1141 { The~key~'#1'~requires~implementation~details~\msg_line_context:. }
1142 {
1143   \c_msg_coding_error_text_tl
1144   When~creating~template~code~using~\DeclareTemplateCode,~
1145   each~key~name~must~have~an~associated~implementation.\\
1146   This~should~be~given~after~a~'= '~sign:~LaTeX~did~not~find~one.
1147 }
1148 \msg_new:nnnn { xtemplate } { key-not-implemented }
1149 { Key~'#1'~has~no~implementation~\msg_line_context:. }
1150 {
1151   \c_msg_coding_error_text_tl
1152   The~definition~of~key~implementations~for~template~'#2'~
1153   of~object~type~'#3'~does~not~include~any~details~for~key~'#1'.\\
1154   The~key~was~declared~in~the~interface~definition,~
1155   and~so~an~implementation~is~required.
1156 }
1157 \msg_new:nnnn { xtemplate } { missing-keytype }
1158 { The~key~'#1'~is~missing~a~key~type~\msg_line_context:. }
1159 {
1160   \c_msg_coding_error_text_tl
1161   Key~interface~definitions~should~be~of~the~form\\

```

```

1162 \ \ #1~::~<key-type>\ \
1163 but~LaTeX~could~not~find~a~<key-type>.
1164 }
1165 \msg_new:nnnn { xtemplate } { no-template-code }
1166 {
1167   The~template~'#2'~of~type~'#1'~is~unknown~
1168   or~has~no~implementation.
1169 }
1170 {
1171   \c_msg_coding_error_text_tl
1172   There~is~no~code~available~for~the~template~name~given.\ \
1173   This~should~be~given~using~\DeclareTemplateCode.
1174 }
1175 \msg_new:nnnn { xtemplate } { object-type-mismatch }
1176 { Object~types~'#1'~and~'#2'~do~not~agree. }
1177 {
1178   You~are~trying~to~use~a~template~directly~with~\UseInstance
1179   (or~a~similar~function),~but~the~object~types~do~not~match.
1180 }
1181 \msg_new:nnnn { xtemplate } { unknown-attribute }
1182 { The~template~attribute~'#1'~is~unknown. }
1183 {
1184   There~is~a~definition~in~the~current~template~reading\ \
1185   \ \ \token_to_str:N \KeyValue {~#1~} \ \
1186   but~there~is~no~key~called~'#1'.
1187 }
1188 \msg_new:nnnn { xtemplate } { unknown-choice }
1189 { The~choice~'#2'~was~not~declared~for~key~'#1'. }
1190 {
1191   The~key~'#1'~takes~a~fixed~list~of~choices~
1192   and~this~list~does~not~include~'#2'.
1193 }
1194 \msg_new:nnnn { xtemplate } { unknown-default-choice }
1195 { The~default~choice~'#2'~was~not~declared~for~key~'#1'. }
1196 {
1197   The~key~'#1'~takes~a~fixed~list~of~choices~
1198   and~this~list~does~not~include~'#2'.
1199 }
1200 \msg_new:nnnn { xtemplate } { unknown-instance }
1201 { The~instance~'#2'~of~type~'#1'~is~unknown. }
1202 {
1203   You~have~asked~to~use~an~instance~'#2',~
1204   but~this~has~not~been~created.
1205 }
1206 \msg_new:nnnn { xtemplate } { unknown-key }
1207 { Unknown~template~key~'#1'. }
1208 {
1209   \c_msg_coding_error_text_tl
1210   The~key~'#1'~was~not~declared~in~the~interface~
1211   for~the~current~template.

```

```

1212 }
1213 \msg_new:nnnn { xtemplate } { unknown-keytype }
1214 { The~key~type~'#1'~is~unknown. }
1215 {
1216   \c_msg_coding_error_text_tl
1217   Valid~key~types~are:\\
1218   --boolean;\\
1219   --choice;\\
1220   --code;\\
1221   --commalist;\\
1222   --function;\\
1223   --instance;\\
1224   --integer;\\
1225   --length;\\
1226   --muskip;\\
1227   --real;\\
1228   --skip;\\
1229   --tokenlist.
1230 }
1231 \msg_new:nnnn { xtemplate } { unknown-object-type }
1232 { The~object~type~'#1'~is~unknown. }
1233 {
1234   \c_msg_coding_error_text_tl
1235   An~object~type~needs~to~be~declared~with~\DeclareObjectType
1236   prior~to~using~it.
1237 }
1238 \msg_new:nnnn { xtemplate } { unknown-template }
1239 { The~template~'#2'~of~type~'#1'~is~unknown. }
1240 {
1241   No~interface~has~been~declared~for~a~template~
1242   '#2'~of~object~type~'#1'.
1243 }

```

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

```

1244 \msg_new:nnn { xtemplate } { declare-object-type }
1245 { Declaring~object~type~'#1'~taking~#2~argument(s)~\msg_line_context:. }
1246 \msg_new:nnn { xtemplate } { declare-template-code }
1247 { Declaring~code~for~template~'#2'~of~object~type~'#1'~\msg_line_context:. }
1248 \msg_new:nnn { xtemplate } { declare-template-interface }
1249 {
1250   Declaring~interface~for~template~'#2'~of~object~type~'#1'~
1251   \msg_line_context:.
1252 }
1253 \msg_new:nnn { xtemplate } { show-no-attribute }
1254 { The~template~'#2'~of~object~type~'#1'~has~no~#3 . }
1255 \msg_new:nnn { xtemplate } { show-attribute }
1256 { The~template~'#2'~of~object~type~'#1'~has~#3 : }
1257 \msg_new:nnn { xtemplate } { show-no-values }
1258 {
1259   The~ \tl_if_empty:nF {#2} {collection~} instance~'#3'~

```

```

1260     \tl_if_empty:nF {#2} { (from-collection~'~{#2}')~ }
1261     of-object-type~'~{#1}'~has-no-values.
1262   }
1263   \msg_new:nnn { xtemplate } { show-values }
1264   {
1265     The~ \tl_if_empty:nF {#2} {collection~} instance~'~{#3}'~
1266     \tl_if_empty:nF {#2} { (from-collection~'~{#2}')~ }
1267     of-object-type~'~{#1}'~
1268     \str_if_eq:nnF { \q_no_value } {#4} { (from-template~'~{#4}')~ }
1269     has-values:
1270   }

```

12.16 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.

```

\DeclareObjectType \DeclareTemplateInterface \DeclareTemplateCode
\DeclareRestrictedTemplate \EditTemplateDefaults \DeclareInstance
\DeclareCollectionInstance \EditInstance \EditCollectionInstance
\UseTemplate \UseInstance \UseCollection
1271 \cs_new_protected_nopar:Npn \DeclareObjectType #1#2
1272   { \__xtemplate_declare_object_type:nn {#1} {#2} }
1273 \cs_new_protected:Npn \DeclareTemplateInterface #1#2#3#4
1274   { \__xtemplate_declare_template_keys:nnnn {#1} {#2} {#3} {#4} }
1275 \cs_new_protected:Npn \DeclareTemplateCode #1#2#3#4#5
1276   { \__xtemplate_declare_template_code:nnnnn {#1} {#2} {#3} {#4} {#5} }
1277 \cs_new_protected:Npn \DeclareRestrictedTemplate #1#2#3#4
1278   { \__xtemplate_declare_restricted:nnnn {#1} {#2} {#3} {#4} }
1279 \cs_new_protected:Npn \DeclareInstance #1#2#3#4
1280   { \__xtemplate_declare_instance:nnnnn {#1} {#3} { } {#2} {#4} }
1281 \cs_new_protected:Npn \DeclareCollectionInstance #1#2#3#4#5
1282   { \__xtemplate_declare_instance:nnnnn {#2} {#4} {#1} {#3} {#5} }
1283 \cs_new_protected:Npn \EditTemplateDefaults #1#2#3
1284   { \__xtemplate_edit_defaults:nnn {#1} {#2} {#3} }
1285 \cs_new_protected:Npn \EditInstance #1#2#3
1286   { \__xtemplate_edit_instance:nnnn {#1} { } {#2} {#3} }
1287 \cs_new_protected:Npn \EditCollectionInstance #1#2#3#4
1288   { \__xtemplate_edit_instance:nnnn {#2} {#1} {#3} {#4} }
1289 \cs_new_protected_nopar:Npn \UseTemplate #1#2#3
1290   { \__xtemplate_use_template:nnn {#1} {#2} {#3} }
1291 \cs_new_protected_nopar:Npn \UseInstance #1#2
1292   { \__xtemplate_use_instance:nn {#1} {#2} }
1293 \cs_new_protected_nopar:Npn \UseCollection #1#2
1294   { \__xtemplate_use_collection:nn {#1} {#2} }

```

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

```

\ShowTemplateCode \ShowTemplateDefaults \ShowTemplateInterface
\ShowTemplateVariables \ShowInstanceValues \ShowCollectionInstanceValues
1295 \cs_new_protected_nopar:Npn \ShowTemplateCode #1#2
1296   { \__xtemplate_show_code:nn {#1} {#2} }
1297 \cs_new_protected_nopar:Npn \ShowTemplateDefaults #1#2
1298   { \__xtemplate_show_defaults:nn {#1} {#2} }

```


_xtemplate_assign_choice: ...	800, 800	_xtemplate_declare_template_code:nnnnn	405, 405, 1276
_xtemplate_assign_choice_aux:n ..	800	_xtemplate_declare_template_keys:nnnn	192, 192, 1274
_xtemplate_assign_choice_aux:nF	817, 825	_xtemplate_edit_defaults:nnn
_xtemplate_assign_choice_aux:o ..	800	602, 602, 1284	_xtemplate_edit_defaults_aux:nnn
_xtemplate_assign_choice_aux:xF	802, 805	599, 602, 605, 607
_xtemplate_assign_code:	826, 826	_xtemplate_edit_instance:nnnn	699, 699, 1286, 1288
_xtemplate_assign_code:n	826, 830, 832, 835	_xtemplate_edit_instance_aux:nnnnn	699, 714, 719
_xtemplate_assign_commalist:	926, 926	_xtemplate_edit_instance_aux:nonnn	699, 706
_xtemplate_assign_function: ..	836, 836	_xtemplate_execute_if_arg_agree:nnT	44, 44, 196, 409
_xtemplate_assign_function_aux:N	836, 839, 840, 842	_xtemplate_execute_if_code_exist:nnT
_xtemplate_assign_instance: ..	853, 853	54, 54, 669, 765, 1032	_xtemplate_execute_if_keys_exist:nnT
_xtemplate_assign_instance_aux:N	853, 856, 857, 859	76
_xtemplate_assign_integer: ..	871, 871	_xtemplate_execute_if_keytype_exist:nT	63, 63, 69
_xtemplate_assign_length: ...	871, 877	_xtemplate_execute_if_keytype_exist:oT	63, 213
_xtemplate_assign_muskip: ...	871, 883	_xtemplate_execute_if_type_exist:nT	70, 70, 194, 407
_xtemplate_assign_real:	871, 889	_xtemplate_find_global: ..	741, 750, 750	_xtemplate_find_global_aux:w
_xtemplate_assign_skip:	871, 895	750, 755, 758	_xtemplate_get_collection:n
_xtemplate_assign_tokenlist:	901, 901	_xtemplate_if_eval_now:n	94
_xtemplate_assign_tokenlist_aux:NN	_xtemplate_if_eval_now:nTF	94, 333, 359, 371, 383, 395
.....	901, 904, 905, 907, 929, 930	_xtemplate_if_instance_exist:nnn	100	_xtemplate_if_instance_exist:nnnF
_xtemplate_assign_variable:N	691, 1312	_xtemplate_if_instance_exist:nnnT
.....	874, 875, 880, 881,	_xtemplate_if_instance_exist:nnnTF	100, 701, 981, 991, 1308
.....	886, 887, 892, 893, 898, 899, 932, 932	_xtemplate_if_key_value:n	85
_xtemplate_assignments_pop:	1007, 1007, 1316	_xtemplate_if_key_value:nF	92
_xtemplate_assignments_push:n	_xtemplate_if_key_value:nT	91
.....	687, 1008, 1008	_xtemplate_if_key_value:nTF	85, 93
_xtemplate_convert_to_assignments:	_xtemplate_if_key_value:oT	934
.....	684, 720, 720, 770	_xtemplate_if_key_value:oTF	85, 782, 909
_xtemplate_convert_to_assignments_aux:n	_xtemplate_if_keys_exist:nnT	76, 411, 595, 609, 1014, 1023
.....	720, 724, 726		
_xtemplate_convert_to_assignments_aux:nn			
.....	720, 731, 749			
_xtemplate_convert_to_assignments_aux:no			
.....	720, 729			
_xtemplate_create_variable:N			
.....	468, 490, 504, 504			
_xtemplate_declare_instance:nnnnn			
.....	667, 667, 1280, 1282			
_xtemplate_declare_instance_aux:nnnnn			
.....	667, 673, 676, 717			
_xtemplate_declare_object_type:nn			
.....	173, 173, 1272			
_xtemplate_declare_restricted:nnnn			
.....	593, 593, 1278			

_xtemplate_if_use_template:n ...	107	_xtemplate_recover_vars:n	
_xtemplate_if_use_template:nTF 146 , 168 , 662 , 672 , 716 , 768 , 1034	
.....	107 , 968	_xtemplate_set_template_eq:nn	
_xtemplate_implement_choice_elt:n		597 , 656 , 656
.....	524 , 559 , 559 , 588	_xtemplate_show:Nnnn	
_xtemplate_implement_choice_elt:nn		1012 , 1017 , 1026 , 1035 , 1039
.....	524 , 559 , 586	_xtemplate_show_code:nn	1010 , 1010 , 1296
_xtemplate_implement_choices:n ...		_xtemplate_show_defaults:nn	
.....	457 , 519 , 519	1012 , 1012 , 1298
_xtemplate_implement_choices_default:		_xtemplate_show_keytypes:nn	
.....	519 , 528 , 538	1012 , 1021 , 1300
_xtemplate_key_to_value:		_xtemplate_show_values:nnn	
.....	784 , 911 , 935 , 942 , 942	1047 , 1047 , 1304 , 1306
_xtemplate_key_to_value_auxi:w ...		_xtemplate_show_vars:nn	1012 , 1030 , 1302
.....	942 , 943 , 944	_xtemplate_split_keytype:n	
_xtemplate_key_to_value_auxii:w	210 , 266 , 272
.....	942 , 953 , 961	_xtemplate_split_keytype_arg:n ...	
_xtemplate_parse_keys_elt:n	300 , 304 , 304 , 328 , 737
.....	202 , 208 , 208 , 263	_xtemplate_split_keytype_arg:o ...	
_xtemplate_parse_keys_elt:nn	304 , 448 , 550 , 567 , 579 , 653 , 810
.....	202 , 261 , 261	_xtemplate_split_keytype_arg_aux:n	
_xtemplate_parse_keys_elt_aux:	304 , 308 , 326 , 329
.....	208 , 226 , 244	_xtemplate_split_keytype_arg_aux:w	
_xtemplate_parse_keys_elt_aux:n	304 , 312 , 322 , 330
.....	208 , 216 , 231	_xtemplate_split_keytype_aux:w ...	
_xtemplate_parse_values:nn	266 , 281 , 289 , 295
.....	613 , 618 , 618 , 679 , 769	_xtemplate_store_defaults:n	
_xtemplate_parse_values_elt:n	113 , 113 , 203 , 614 , 659
.....	623 , 625 , 625	_xtemplate_store_key_implementation:nnn	
_xtemplate_parse_values_elt:nn	413 , 421 , 421
.....	623 , 630 , 630	_xtemplate_store_keytypes:n	
_xtemplate_parse_values_elt_aux:n		113 , 119 , 204 , 661
.....	630 , 641 , 643 , 650	_xtemplate_store_restrictions:n ..	
_xtemplate_parse_vars_elt:n	113 , 134 , 430 , 615
.....	427 , 437 , 437	_xtemplate_store_value_boolean:n .	
_xtemplate_parse_vars_elt:nn	331 , 331
.....	427 , 439 , 439	_xtemplate_store_value_choice:n ..	
_xtemplate_parse_vars_elt_aux:n	349 , 351
.....	449 , 454 , 454	_xtemplate_store_value_code:n	
_xtemplate_parse_vars_elt_aux:w	349 , 349 , 351 , 352 , 353 , 354 , 355 , 356
.....	454 , 474 , 483	_xtemplate_store_value_commalist:n	
_xtemplate_recover_defaults:n	349 , 352
.....	146 , 146 , 423 , 611 , 658 , 671 , 767 , 1016	_xtemplate_store_value_function:n	
_xtemplate_recover_keytypes:n	349 , 353
.....	146 , 151 , 424 , 620 , 660 , 1025	_xtemplate_store_value_instance:n	
_xtemplate_recover_restrictions:n		349 , 354
.....	146 , 158 , 612	_xtemplate_store_value_integer:n .	
_xtemplate_recover_values:n	357 , 357
.....	146 , 163 , 703 , 1051	_xtemplate_store_value_length:n ..	
		357 , 369

__xtemplate_store_value_muskip:n
...	357, 381	__xtemplate_restrict_root_tl
__xtemplate_store_value_real:n	349, 355	...
__xtemplate_store_value_skip:n	357, 393	...
__xtemplate_store_value_tokenlist:n
...	349, 356	...
__xtemplate_store_values:n	113, 128, 683	...
__xtemplate_store_vars:n
...	113, 140, 428, 663	...
__xtemplate_use_collection:nn
...	999, 999, 1294	...
__xtemplate_use_instance:nn
...	865, 966, 966, 1292	...
__xtemplate_use_instance_aux:nNnnn
...	966, 969, 972	...
__xtemplate_use_instance_aux:nn
...	966, 970, 978	...
__xtemplate_use_template:nnn
...	763, 763, 975, 1290	...
...	1121, 1129, 1138, 1162, 1185	...
A		
AssignTemplateKeys
B		
bool_if:cTF
bool_if:NF
bool_if:NTF
...	855, 873, 879, 885, 891, 897, 903, 928	...
bool_if:nTF
bool_new:N
bool_set_false:N
bool_set_true:N
C		
c__xtemplate_code_root_tl
...	5, 56, 415, 664, 665, 689, 771, 1011	...
c__xtemplate_defaults_root_tl
...	5, 6, 115, 116, 149	...
c__xtemplate_instances_root_tl
...	5, 7, 103, 685, 694, 695, 986, 992	...
c__xtemplate_key_order_root_tl
...	5, 9, 124, 125, 156	...
c__xtemplate_keytypes_arg_seq
...	13, 13, 14, 15, 16, 215, 325	...
c__xtemplate_keytypes_root_tl
...	5, 8, 78, 121, 122, 154	...
...	5, 10, 136, 137, 161	...
...	5, 11, 130, 131, 166	...
...	5, 12, 142, 143, 171	...
...	1074, 1083, 1105, 1113, 1127, 1135, 1143, 1151, 1160, 1171, 1209, 1216, 1234	...
...	178	...
...	541, 545, 590, 803, 806	...
...	179	...
...	268	...
...	267	...
...	429, 621	...
...	136	...
...	929	...
...	929	...
...	137	...
...	529	...
...	561	...
...	257	...
...	574	...
...	639	...
...	531	...
...	25, 35, 509	...
...	652	...
...	575	...
...	930	...
...	930	...
...	160	...
...	521	...
...	414	...
...	846	...
...	41, 42, 43, 69, 91, 92, 93, 328, 719, 749, 825	...
...	839	...
...	664	...
...	416, 856	...
...	56, 65, 78, 103	...
...	467, 489	...
...	510	...
...
...	351, 352, 353, 354, 355, 356, 1317	...
...	244, 329, 330, 1007, 1307, 1309, 1311	...
...	44, 54, 63, 70, 76, 113, 119, 128, 134, 140, 146, 151, 158, 163, 168, 173, 192, 208, 261, 272, 289, 304, 331, 349, 357, 369,	...

\l__xtemplate_keytype_arg_tl	\msg_error:nnxxx 50, 433, 553, 568, 580, 811
.. 26, 28, 235, 249, 250, 257, 307,	\msg_info:nnxx
318, 521, 555, 570, 582, 813, 849, 866	\msg_line_context:
\l__xtemplate_keytype_tl 1111, 1118, 1125, 1133,
..... 26, 27, 213, 233,	1141, 1149, 1158, 1245, 1247, 1251
248, 255, 264, 306, 317, 446, 448,	\msg_new:nnn
456, 459, 506, 517, 654, 738, 740, 744	1246, 1248, 1253, 1255, 1257, 1263
\l__xtemplate_keytypes_prop 31,	\msg_new:nnnn . 1071, 1080, 1087, 1096,
31, 123, 153, 199, 252, 431, 444,	1102, 1110, 1117, 1124, 1132, 1140,
450, 548, 565, 577, 634, 728, 808, 1026	1148, 1157, 1165, 1175, 1181, 1188,
\l__xtemplate_restrict_bool	1194, 1200, 1206, 1213, 1231, 1238
..... 24, 24, 598, 604, 637	\muskip_gset:Nn
\l__xtemplate_restrict_clist	\muskip_new:N
.. 25, 25, 138, 160, 429, 621, 639, 652	\muskip_set:Nn
\l__xtemplate_tmp_clist	385, 887
.. 35, 35, 521, 529, 531, 561, 574, 575	
\l__xtemplate_tmp_dim .. 35, 36, 373, 375	P
\l__xtemplate_tmp_int	\prg_new_conditional:Npnn 85, 94, 100, 107
..... 35, 37, 175, 183, 189, 361, 363	\prg_return_false:
\l__xtemplate_tmp_muskip 35, 38, 385, 387	89, 98, 105, 111
\l__xtemplate_tmp_skip . 35, 39, 397, 399	\prg_return_true:
\l__xtemplate_tmp_tl	88, 97, 104, 110
..... 40, 40, 46, 47, 51, 246, 253,	\prop_clear:N
275, 276, 277, 278, 282, 527, 540,	198, 199, 425
541, 542, 544, 545, 546, 549, 550,	\prop_clear_new:c
552, 566, 567, 578, 579, 589, 591,	130
635, 653, 705, 706, 728, 729, 809,	\prop_gclear_new:c
810, 822, 823, 946, 950, 958, 1060, 1062	115, 121, 142
\l__xtemplate_value_tl	\prop_get:NnN
..... 26, 29, 733, 782, 789,	46, 704, 728
795, 803, 812, 832, 850, 867, 909,	\prop_get:NnNF
915, 922, 934, 939, 943, 951, 953, 964	43, 1003
\l__xtemplate_values_prop	\prop_get:NnNT
.. 31, 33, 117, 132, 148, 165, 198,	42, 733
337, 341, 346, 350, 362, 366, 374,	\prop_get:NnNTF
378, 386, 390, 398, 402, 526, 551,	41, 735, 819
682, 704, 733, 1017, 1052, 1059, 1065	\prop_get:NoN 548, 551, 565, 577, 808
\l__xtemplate_var_tl	\prop_get:NoNT
..... 26, 30, 735, 753, 755, 760,	526
788, 796, 831, 847, 863, 914, 921, 938	\prop_get:NoNTF
\l__xtemplate_vars_prop	41, 443, 634, 948
.. 31, 34, 144, 170, 425, 461, 469, 491,	\prop_gput:NnV
522, 542, 546, 591, 735, 820, 949, 1035	188
	\prop_gset_eq:cN
M	116, 122, 143
\msg_error:nn	\prop_if_empty:NTF
258	1042, 1052
\msg_error:nnx	\prop_if_in:NnTF
74, 222, 237, 286, 299, 438, 452,	72
476, 495, 500, 533, 628, 646, 746, 957	\prop_if_in:NoF
\msg_error:nnxx 59, 81, 182, 710, 976, 994	542, 546
	\prop_map_function:NN
	1045, 1065
	\prop_map_inline:Nn
	431
	\prop_new:N
	17, 20, 31, 33, 34
	\prop_pop:NnN
	1059
	\prop_put:Nnn
	682, 1000
	\prop_put:Non . 337, 341, 346, 350, 366,
	378, 390, 402, 461, 469, 491, 522, 591
	\prop_put:Noo
	252
	\prop_put:NVV
	362, 374, 386, 398
	\prop_remove:NV
	450
	\prop_set_eq:cN
	131
	\prop_set_eq:Nc
	148, 153, 165, 170
	\ProvidesExplPackage
	3
	Q
	\q_mark
	954

\q_nil	87, 96, 109, 954		
\q_no_value	1268		
\q_stop .	87, 96, 109, 282, 289, 295, 313,		
	322, 330, 474, 483, 755, 758, 954, 961		
\quark_if_nil:NF	963		
S			
\seq_clear:N	200		
\seq_gclear_new:c	124		
\seq_gset_eq:cN	125		
\seq_if_in:NoTF	219		
\seq_map_break:	240, 319		
\seq_map_function:NN	215, 325, 723		
\seq_new:N	13, 32		
\seq_put_right:Nn	14, 15, 16		
\seq_put_right:No	254		
\seq_set_eq:Nc	155		
\ShowCollectionInstanceValues			
	10, 1295, 1305		
\ShowInstanceValues	9, 1295, 1303		
\ShowTemplateCode	9, 1295, 1295		
\ShowTemplateDefaults	9, 1295, 1297		
\ShowTemplateInterface	9, 1295, 1299, 1317		
\ShowTemplateKeytypes	1317		
\ShowTemplateVariables ..	10, 1295, 1301		
\skip_gset:Nn	898		
\skip_new:N	39		
\skip_set:Nn	397, 899		
\str_case:onn	506		
\str_if_eq:nnF	563, 1268		
\str_if_eq:nnTF	974		
\str_if_eq:noTF	87, 96, 109		
\str_if_eq:onF	738, 740		
\str_if_eq:onT	233, 255		
\str_if_eq:onTF	456, 459		
T			
\tl_clear:N	280, 307, 722, 1005		
\tl_const:Nn	5, 6, 7, 8, 9, 10, 11, 12		
\tl_gset:Nn	904		
\tl_gset:NV	904		
\tl_head:w	87, 96, 109		
\tl_if_empty:NF	249		
\tl_if_empty:nF ..	1259, 1260, 1265, 1266		
\tl_if_empty:NT	235		
\tl_if_empty:nT	315		
\tl_if_empty:NTF	298		
\tl_if_empty:nTF	485		
\tl_if_in:nnT	310		
\tl_if_in:nnTF	292, 473		
\tl_if_in:onT	753		
\tl_if_in:onTF	278		
\tl_if_single:nTF	465, 487		
\tl_new:N	18, 19, 21, 26, 27, 28, 29, 30, 40, 515		
\tl_put_right:Nn	294		
\tl_put_right:No	823		
\tl_put_right:Nx	291,		
	785, 793, 828, 844, 861, 912, 919, 936		
\tl_remove_all:Nn	276, 442, 633, 947		
\tl_replace_all:Nnn	277		
\tl_set:Nn	275,		
	306, 317, 318, 743, 760, 905, 964, 1009		
\tl_set:NV	905		
\tl_set:Nx	246, 441, 540, 544, 589, 632, 946		
\tl_to_lowercase:n	269		
\tl_to_str:n	291, 441, 477, 496, 501, 632, 946		
\token_to_str:N	1185		
U			
\use:c ...	264, 517, 654, 744, 771, 984, 992		
\UseCollection	1271, 1293		
\UseInstance	8, 1178, 1271, 1291		
\UseTemplate	8, 109, 1271, 1289		