

Templates: Prototype document functions^{*}

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

October 30, 2026

1 Introduction

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

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

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

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

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

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

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

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

^{*}This file has version v1.0o dated 2026-10-29, © L^AT_EX Project.

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

2 What is a document?

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

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

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

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

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

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

3 Types, templates, and instances

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

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

4 Template types

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

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

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

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

```
\NewTemplateType{sectioning}{3}
```

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

5 Templates

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

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

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

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

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

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

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

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

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

`\KeyValue` `\KeyValue {⟨key name⟩}`

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

```

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

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

Table 2: Bindings required for different key types when defining template implementations with `\DeclareTemplateCode`. Apart from `choice` and `function` all of these accept the keyword `global` to carry out a global assignment. Apart from `choice`, all of these accept the keyword `name` to allow passing command name without the leading backslash: this is useful for auto-generated names.

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

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

$$\begin{aligned}
\langle key1 \rangle &= \langle variable1 \rangle, \\
\langle key2 \rangle &= \langle variable2 \rangle, \\
\langle key3 \rangle &= \text{global } \langle variable3 \rangle, \\
\langle key4 \rangle &= \text{global } \langle variable4 \rangle, \\
\langle key5 \rangle &= \text{name } \{ \langle name5 \rangle \}, \\
&\dots
\end{aligned}$$

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

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

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

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

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

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

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

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

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

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

to process this key/value list inside the template.

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

`\SetCurrentKnownTemplateKeys` works like `\SetKnownTemplateKeys` but obtains the name of the current `{\type}`/`{\template}` automatically based on the current context. It should be used before any other template code may be called (and thus change the value of the tracking of the current template).

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

For example, a list, such as `enumerate`, is made up from a `blockenv`, `block`, `list`,

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

and a `para` template and in the single user-supplied optional argument of `enumerate` key/values for any of these templates might be specified.

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

On the other hand you may want to allow for “misspellings” without generating an error or a warning. For example, if you define a template that accepts only a few keys, you might just want to ignore anything specified in the source when you use this template in place of a different one, without the need to alter the document source. Or you might just generate a warning message, which is easy, given that the unused key/values are available in the `\UnusedTemplateKeys` variable.

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

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

6 Multiple choices

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

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

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

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

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

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

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

```

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

```

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

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

7 Instances

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

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

`\DeclareInstance` `\DeclareInstance`

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

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

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

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

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

`\InstanceValue` ★ `\InstanceValue {\langle type \rangle} {\langle instance \rangle} {\langle key \rangle}`

Expands to the current value for the $\langle key \rangle$ stored in the $\langle instance \rangle$ of $\langle type \rangle$. If the $\langle instance \rangle$ or $\langle key \rangle$ do not exist, the expansion is empty. The result is returned within the `\unexpanded` primitive (`\exp_not:n`).

`\IfInstanceExistsT` `\IfInstanceExistsTF {\langle type \rangle} {\langle instance \rangle} {\langle true code \rangle} {\langle false code \rangle}`

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

`\DeclareInstanceCopy` `\DeclareInstanceCopy`

`{\langle type \rangle} {\langle instance2 \rangle} {\langle instance1 \rangle}`

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

8 Document interface

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

`\UseInstance` `\UseInstance`

`{\langle type \rangle} {\langle instance \rangle} \langle arguments \rangle`

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

<code>\UseTemplate</code>	<code>\UseTemplate {<type>} {<template>}</code> <code>{<settings>} <arguments></code>
---------------------------	--

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

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

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

9 Changing existing definitions

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

<code>\EditTemplateDefaults</code>	<code>\EditTemplateDefaults</code> <code>{<type>} {<template>} {<new defaults>}</code>
------------------------------------	---

Edits the *<defaults>* for a *<template>* for an *<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.

<code>\EditInstance</code>	<code>\EditInstance</code> <code>{<type>} {<instance>} {<new values>}</code>
----------------------------	---

Edits the *<values>* for an *<instance>* for an *<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.

9.1 Expanding the values of keys

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

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

10 Creating equivalent keys

There are times when it is useful to have different names which map to the same underlying key interfaces, for example where a more complex higher level name is implemented by several simpler underlying keys, or where a key name needs to be changed. To support this, it is possible to define equivalent keys using `\DeclareTemplateInterface`.

```
\DeclareTemplateInterfaceEquivalents \DeclareTemplateInterfaceEquivalents
    {\type} {\template} [{\deprecation-date}] {\key list}
```

Defines one or more keys to act as equivalents to the base keys of the `\template`. As for `\DeclareTemplateInterface`, these take a specialized format:

```
\key1 : \key type1 = \definition1 ,
\key2 : \key type2 = \definition2 ,
...
```

The `\key type` of the key being created should be one of

- **alias**: A direct equivalence between the key being created and a single parent
- **meta**: An equivalence to one or more parent keys, where additional code may be included as well as the value passed to the key being created; the value (provided in the instance declaration or in the ad hoc setting in the document) is available as `#1`.

for example

```
\DeclareTemplateInterfaceEquivalents { type } { template }
{
  alias-key-one : alias = base-key-one ,
  alias-key-two : meta =
    { parent-key-two = {#1}, base-key-three = \hspace{#1} }
}
```

Equivalent key names may be reused to overwrite existing equivalent keys but cannot be the same as base keys of the template.

Optionally, the `\deprecation date` may be set. Before this date, a warning will be printed concerning the keys; after this date, an error will be issued. In both cases, the keys will still be defined.

10.1 Modifying existing key values

Some changes to key values are relative adjustments to the previous value, for example to add 1 cm to a margin. To support this, the syntax

```
key-a += 1cm
```

can be used, with a `+` immediately before the `=` sign. This will cause the new value to be *added* to the existing one. This naturally applies to all numeric key types, and for `tokenlist` and `commalist` means adding the new material to the end of the existing value. An error will arise if this is used for other key types, for example `function`.

11 Getting information about templates and instances

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

```
Template 'example' of type 'example' has variable mapping:  
> demo unknown => \def \demo {?}  
> demo c      => \def \demo {c}  
> demo b      => \def \demo {b}  
> demo a      => \def \demo {a}.
```

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

12 Debugging support

<hr/> <hr/>	<code>\DebugTemplatesOn</code>
<code>\DebugTemplatesOff</code>	Turn on or off debugging support for use of templates. Debugging information is provided at the point of <i>use</i> of templates and instances, i.e. where normal messages should be avoided for performance reasons.

Index

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

A		int commands:	
\AssignTemplateKeys	6	\l_tmpa_int	5
B		\item 7	
bool commands:		K	
\l_tmpa_bool	5	\KeyValue	4
C		M	
\caption	2	muskip commands:	
clist commands:		\l_tmpa_muskip	5
\l_tmpa_clist	5	N	
D		\NewTemplateType	3
\DebugTemplatesOff	12	S	
\DebugTemplatesOn	12	\section	2
\DeclareInstance	9	\SetCurrentKnownTemplateKeys	6
\DeclareInstanceCopy	9	\SetKnownTemplateKeys	6
\DeclareTemplateCode	3	\SetTemplateKeys	6
\DeclareTemplateCopy	7	\ShowInstanceValues	12
\DeclareTemplateInterface	11	\ShowTemplateCode	12
\DeclareTemplateInterfaceEquivalents	11	\ShowTemplateDefaults	12
dim commands:		\ShowTemplateInterface	12
\l_tmpa_dim	5	\ShowTemplateVariables	12
E		skip commands:	
\EditInstance	10	\l_tmpa_skip	5
\EditTemplateDefaults	10	T	
F		tl commands:	
fp commands:		\l_tmpa_tl	5
\l_tmpa_fp	5	U	
I		\UnusedTemplateKeys	6
\IfInstanceExistsF	9	use commands:	
\IfInstanceExistsT	9	\use_i:nn	5
\IfInstanceExistsTF	9	\UseInstance	9
\InstanceValue	9	\UseTemplate	10