The `lttemplates.dtx` code*

Frank Mittelbach, Chris Rowley, David Carlisle, LaTeX Project†

February 22, 2025

1 Introduction

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

1. authoring of the text with mark-up;
2. document layout design;
3. implementation (with TeX 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 TeX implementation in the middle is the glue between the two.

LaTeX'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 LaTeX 2_ε classes look somewhat dated now in terms of their visual design, their typography is generally sound (barring the occasional minor faults).

However, LaTeX 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 LaTeX programmers to provide their own customizations on top of a pre-existing class.

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

†E-mail: latex-team@latex-project.org

2 What is a document?

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

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

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

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

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

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

3 Types, templates, and instances

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

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

4 Template types

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

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

`\NewTemplateType` \NewTemplateType {⟨*template type*⟩} {⟨*no. of args*⟩}

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`.

`\DeclareTemplateInterface` \DeclareTemplateInterface
{⟨*type*⟩} {⟨*template*⟩} {⟨*no. of args*⟩}
{⟨*key list*⟩}

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

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

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

Key-type	Description of input
`boolean`	`true` or `false`
`choice{⟨choices⟩}`	A list of pre-defined ⟨*choices*⟩
`commalist`	A comma-separated list
`function{⟨N⟩}`	A function definition with N arguments (N from 0 to 9)
`instance{⟨name⟩}`	An instance of type ⟨*name*⟩
`integer`	An integer or integer expression
`length`	A fixed length
`muskip`	A math length with shrink and stretch components
`real`	A real (floating point) value
`skip`	A length with shrink and stretch components
`tokenlist`	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
`boolean`	Boolean variable, *e.g.* `\l_tmpa_bool`
`choice`	List of choice implementations (see Section 6)
`commalist`	Comma list, *e.g.* `\l_tmpa_clist`
`function`	Function taking N arguments, *e.g.* `\use_i:nn`
`instance`	
`integer`	Integer variable, *e.g.* `\l_tmpa_int`
`length`	Dimension variable, *e.g.* `\l_tmpa_dim`
`muskip`	Muskip variable, *e.g.* `\l_tmpa_muskip`
`real`	Floating-point variable, *e.g.* `\l_tmpa_fp`
`skip`	Skip variable, *e.g.* `\l_tmpa_skip`
`tokenlist`	Token list variable, *e.g.* `\l_tmpa_tl`

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.

`\DeclareTemplateCode`

`\DeclareTemplateCode`
  `{⟨type⟩} {⟨template⟩} {⟨no. of args⟩}`
  `{⟨key bindings⟩} {⟨code⟩}`

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

```
⟨key1⟩ = ⟨variable1⟩,
⟨key2⟩ = ⟨variable2⟩,
⟨key3⟩ = global ⟨variable3⟩,
⟨key4⟩ = global ⟨variable4⟩,
...
```

With the exception of the choice, code and function key types, the ⟨*variable*⟩ here should be the name of an existing LaTeX3 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 type.

`\AssignTemplateKeys`

`\AssignTemplateKeys`

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.

`\SetKnownTemplateKeys` `\SetTemplateKeys` `\UnusedTemplateKeys`

```
\SetKnownTemplateKeys {⟨type⟩} {⟨template⟩} {⟨keyvals⟩}
\SetTemplateKeys      {⟨type⟩} {⟨template⟩} {⟨keyvals⟩}
\UnusedTemplateKeys % all ⟨keyvals⟩ unused by previous \SetKnownTemplateKeys
```

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

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

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

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

to process this key/value list inside the template.

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

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

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

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

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

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

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

`\DeclareTemplateCopy`

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

Copies ⟨*template1*⟩ of ⟨*type*⟩ to a new name ⟨*template2*⟩: the copy can then be edited independent of the original.

6 Multiple choices

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

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

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

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

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

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

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

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

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

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

7 Instances

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

For example, a template might say "here is a section with or without a number that might be centered or left aligned and print its contents in a certain font of a certain size, with a bit of a gap before and after it" whereas an instance declares "this is a section with a number, which is centered and set in 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`
`{⟨type⟩} {⟨instance⟩} {⟨template⟩} {⟨parameters⟩}`

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

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

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

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

`\IfInstanceExistsT`
`\IfInstanceExistsF`
`\IfInstanceExistsTF`

`\IfInstanceExistsTF {⟨type⟩} {⟨instance⟩} {⟨true code⟩} {⟨false code⟩}`

Tests if the named ⟨*instance*⟩ of a ⟨*type*⟩ exists, and then inserts the appropriate code into the input stream.

`\DeclareInstanceCopy`

`\DeclareInstanceCopy`
`{⟨type⟩} {⟨instance2⟩} {⟨instance1⟩}`

Copies the ⟨*values*⟩ for ⟨*instance1*⟩ for an ⟨*type*⟩ to ⟨*instance2*⟩.

8 Document interface

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

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

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

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

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

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

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

9 Changing existing definitions

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

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

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.

`\EditInstance` `\EditInstance`
`{⟨type⟩} {⟨instance⟩} {⟨new values⟩}`

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.

10 Getting information about templates and instances

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

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

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

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

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

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

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

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

`\ShowTemplateVariables` `\ShowTemplateVariables {⟨type⟩} {⟨template⟩}`

Shows the ⟨*variables*⟩ and associated ⟨*keys*⟩ of a ⟨*template*⟩ for an ⟨*type*⟩ in the terminal. Note that `code` and `choice` keys do not map directly to variables but to arbitrary code. For `choice` keys, each valid choice is shown as a separate entry in the list, with the key name and choice separated by a space, for example

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

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

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.