The latex-lab-prototype package
Prototype document functions

The LaTeX Project*

Released 2022-03-09

Contents

1 What is a document? 2

2 Object types 3

3 Templates 3

4 Instances 4

5 Document interface 5

6 Showing template information 6

7 Open questions and comparison with xtemplate 6

7.1 Module name . . . 6
7.2 Design-level names . . . 6
7.3 Objects . . . 6
7.4 Efficiency and repetition of key setting . . . 6
7.5 Key ordering . . . 6
7.6 Setting defaults . . . 7
7.7 The need for templates and instances . . . 7
7.8 Assignment of key values . . . 7
7.9 Values from other keys . . . 7
7.10 The nature of debugging data . . . 7
7.11 Collections . . . 7

*E-mail: latex-team@latex-project.org

8 latex-lab-prototype Implementation 8
8.1 File declaration . . . 8
8.2 \keys_precompile:nnN . . . 8
8.3 Setup . . . 10
8.4 Data structures . . . 10
8.5 Creating objects . . . 10
8.6 Templates and instances . . . 11
8.7 Showing information . . . 13
8.8 Messages . . . 13

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 customisation.

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

The idea behind this module 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. The approach here also makes it easier for LATEX programmers to provide their own customisations on top of a pre-existing class.

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 2, 3, and 4, respectively.

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.

2 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.)

`\prototype_declare_object:nn` `\prototype_declare_object:nn {⟨object type⟩} {⟨no. of args⟩}`

This function defines an ⟨*object type*⟩ taking ⟨*no. of arguments*⟩, where the ⟨*object type*⟩ is an abstraction as discussed above. For example,

```
\prototype_declare_object:nn { sectioning } { 3 }
```

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

While not formally verified the semantics of all arguments are part of the object declaration and need to be carefully documented in order to make the use of different templates for the same object type meaningful.

3 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.

`\prototype_declare_template:nnnn` `\prototype_declare_template:nnnn {⟨object type⟩} {⟨template⟩} {⟨key definitions⟩} {⟨code⟩}`

A ⟨*template*⟩ interface is declared for a particular ⟨*object type*⟩. The interface itself is defined by the ⟨*key definitions*⟩, which is itself a key–value list using the same interface as `\keys_define:nn`. (The keys created here *are* managed l3keys in the tree `prototype/`⟨*object*⟩`/`⟨*template*⟩). As described below, the keys should be defined such that they can be set multiple times: first to a default value, then to a specific value for an instance and finally to a per-use override.

The ⟨*code*⟩ argument of `\template_declare_template:nnnn` 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. The template and instance key values (see below) are assigned before the ⟨*code*⟩ is inserted.

`\prototype_declare_defaults:nnn` `\prototype_declare_template:nnnn {⟨object type⟩} {⟨template⟩} {⟨defaults⟩}`

Sets the default values for each ⟨*key*⟩ in a ⟨*template*⟩. When a template is used, these values are applied first *before* those set by `\prototype_use_template:nnn` or `\prototype_declare_instance:nnnn`. If not default is given, the prevailing state when the template is used will apply.

4 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.

`\prototype_declare_instance:nnnn` `\prototype_declare_instance:nnnn {⟨object type⟩} {⟨template⟩} {⟨instance⟩} {⟨parameters⟩}`

This function uses a ⟨*template*⟩ for an ⟨*object 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 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:

```
\prototype_declare_instance:nnnn { sectioning } { basic } { section-num }
  {
    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.

5 Document interface

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

`\prototype_use_instance:nn` `\prototype_use_instance:nn {⟨object type⟩} {⟨instance⟩} ⟨arguments⟩`
`\prototype_use_instance:nnn` `\prototype_use_instance:nnn {⟨object type⟩} {⟨instance⟩} {⟨overrides⟩} ⟨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. The `nnn` version allows for local overrides of the instance settings using the additional keyval argument.

`\prototype_use_template:nnnn` `\prototype_use_template:nnnn {⟨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 `\template_declare_instance:nnnn` and immediately using it with `\template_use_instance:nnn`, but without the instance having any further existence. It is therefore useful where a template needs to be used once.

6 Showing template information

`\prototype_show_template_code:nn`	`\prototype_show_template_code:nn {`⟨*object type*⟩`} {`⟨*template*⟩`}`
`\prototype_show_template_defaults:nn`	`\prototype_show_template_defaults:nn {`⟨*object type*⟩`} {`⟨*template*⟩`}`
`\prototype_show_instance_values:nn`	`\prototype_show_instance_values:nn {`⟨*object type*⟩`} {`⟨*instance*⟩`}`

Show information about a declare template or instance for debugging purposes.

7 Open questions and comparison with xtemplate

The approach here is modelled on that from xtemplate, but since it uses l3keys rather than dedicated key handling, there are some differences. There is also a simplification in that collections are not supported (because we now think that they provided the wrong kind of abstraction).

The various open questions, including those linked to xtemplate concepts, are collected here.

7.1 Module name

This is currently open for ideas: traditionally `template` has been used. This may link to the need for both templates and instances (*vide infra*).

7.2 Design-level names

These are currently not provided. That allows both this code and xtemplate to be loaded in the same document. We will likely want to decide on these names: they could for example include `Prototype` or `Design`, or could use the existing xtemplate if a compatibility approach can be designed.

7.3 Objects

Is this name clear? A possible alternative is 'element'.

7.4 Efficiency and repetition of key setting

In the xtemplate implementation, keys values are stored in property lists before being applied. This means that when creating an instance, the template defaults can be replaced entirely by any instance values. In contrast, the approach here simply precompiles all of the template defaults, then appends the precompiled list from the instance. Some variables are therefore set twice. More importantly, this means that arbitrary code could be executed twice: authors need to be aware of this.

7.5 Key ordering

Linked to the previous idea, in xtemplate keys are set in the order they are declared in the setup. In contrast, using l3keys they are set in the order the keys are given in the input. Is this OK?

7.6 Setting defaults

The current approach requires setting the defaults separately from the key creation. That means listing keys twice. However, it also avoids further overloading of the keyval setup. Is this reasonable?

7.7 The need for templates and instances

In xtemplate, storing keys in a `prop` means that there is a real efficiency when creating an instance. In contrast, using precompiled keys here, creating a template and creating an instance are almost identical. Could we drop the distinction? That would then work well if we allowed instances to be derived from others: effectively the same as instances from templates, but with a 'flatter' approach.

7.8 Assignment of key values

The xtemplate approach uses `\AssignTemplateKeys` to specify when keys are assigned. In contrast, here key assignment is automatic. If you look over TEX Live, the only places that `\AssignTemplateKeys` is not the first thing in the code are in limited use cases in enotez and xgalley. In both packages, that's because they want to limit the scope of assignment. In enotez the key setting is placed inside a group, whereas in xgalley there is a save-and-restore approach as a group is not possible. Both of those use-cases could be covered in other ways: it's a question of setting up the template keys so they assign to an intermediate variable, then assigning those as necessary to the live ones for these cases.

The main reason for not using `\AssignTemplateKeys` is that the common case doesn't need it. We could of course stick to an explicit-assignment approach, or have two variants or template-creation, etc., where assignment is manual in one of them.

7.9 Values from other keys

The xtemplate approach offers `\KeyValue` to pass the value of one key as the default for another. That relies on the fact that key setting is ordered (*vide supra*). It also means that there is some code to check for this as part of key setting: it's non-trivial to support. The current l3keys-based code doesn't offer this. Instead one could use for example meta keys. That is a different interface and might occasionally be awkward. We can add some `.store-value:n` property to allow a `\keys_value:nn` approach, but without key ordering it might still not work in the same way.

7.10 The nature of debugging data

Due to the differences in data storage, the xtemplate method offers a richer ability to debug template internals than the one here. We can look at *e.g.* tracking all keys for a template to make this easier. It is worth noting that much of this data is really something that should be part of the documentation anyway. Also, it would be trivial to save the raw defaults and do the 'hard' processing only if asked to show the values (*i.e.* using code similar to that in xtemplate).

7.11 Collections

These are not implemented at all: we likely want a new approach to contexts.

8 latex-lab-prototype Implementation

8.1 File declaration

```
⟨*package⟩
\ProvidesFile{latex-lab-prototype.sty}
  [2022-03-09 v0.1b Experimental prototype document functions]
⟨/package⟩
⟨*2ekernel⟩
\ExplSyntaxOn
```

8.2 \keys_precompile:nnN

```
⟨@@=keys⟩
```

This may not yet be available in expl3 so we ensure it is set up here: all temporary. We just redefine those internals that need it.

```
\tl_if_exist:NF \l__keys_precompile_tl
  {
    \bool_new:N \l__keys_precompile_bool
    \tl_new:N \l__keys_precompile_tl
  }
\cs_gset_protected:Npn \__keys_precompile:n #1
  {
    \bool_if:NTF \l__keys_precompile_bool
      { \tl_put_right:Nn \l__keys_precompile_tl }
      { \use:n }
        {#1}
  }
\cs_gset_protected:Npn \__keys_bool_set:Nnnn #1#2#3#4
  {
    \bool_if_exist:NF #1 { \bool_new:N #1 }
    \__keys_choice_make:
    \__keys_cmd_set:nx { \l_keys_path_str / true }
      { \exp_not:c { bool_ #2 set_ #3 :N } \exp_not:N #1 }
    \__keys_cmd_set:nx { \l_keys_path_str / false }
      { \exp_not:c { bool_ #2 set_ #4 :N } \exp_not:N #1 }
    \__keys_cmd_set_direct:nn { \l_keys_path_str / unknown }
      {
        \msg_error:nnx { keys } { boolean-values-only }
          \l_keys_key_str
      }
    \__keys_default_set:n { true }
  }
\cs_gset_protected:Npn \__keys_choice_make_aux:N #1
  {
    \cs_set_nopar:cpn { \c__keys_type_root_str \l_keys_path_str }
      { choice }
    \__keys_cmd_set_direct:nn \l_keys_path_str { #1 {##1} }
    \__keys_cmd_set_direct:nn { \l_keys_path_str / unknown }
      {
        \msg_error:nnxx { keys } { choice-unknown }
          \l_keys_path_str {##1}
      }
  }
```

```
\cs_gset_protected:Npn \__keys_cmd_set:nn #1#2
  {  \__keys_cmd_set_direct:nn {#1} { \__keys_precompile:n {#2} } }
\cs_gset_protected:Npn \__keys_cmd_set_direct:nn #1#2
  { \cs_set_protected:cpn { \c__keys_code_root_str #1 } ##1 {#2} }
\cs_gset_protected:Npn \__keys_cs_set:NNpn #1#2#3#
  {
    \cs_set_protected:cpx { \c__keys_code_root_str \l_keys_path_str } ##1
      {
        \__keys_precompile:n
          { #1 \exp_not:N #2 \exp_not:n {#3} {##1} }
      }
    \use_none:n
  }
\cs_gset_protected:Npn \__keys_meta_make:n #1
  {
    \exp_args:NVo \__keys_cmd_set_direct:nn \l_keys_path_str
      {
        \exp_after:wN \keys_set:nn \exp_after:wN
          { \l__keys_module_str } {#1}
      }
  }
\cs_gset_protected:Npn \__keys_meta_make:nn #1#2
  {
    \exp_args:NV \__keys_cmd_set_direct:nn
      \l_keys_path_str { \keys_set:nn {#1} {#2} }
  }
\cs_gset_protected:Npn \keys_precompile:nnN #1#2#3
  {
    \bool_set_true:N \l__keys_precompile_bool
    \tl_clear:N \l__keys_precompile_tl
    \keys_set:nn {#1} {#2}
    \bool_set_false:N \l__keys_precompile_bool
    \tl_set_eq:NN #3 \l__keys_precompile_tl
  }
\cs_gset_protected:Npn \__keys_show:Nnn #1#2#3
  {
    #1 { keys } { show-key }
      { \__keys_trim_spaces:n { #2 / #3 } }
      {
        \keys_if_exist:nnT {#2} {#3}
          {
            \exp_args:Nnf \msg_show_item_unbraced:nn { code }
              {
                \exp_args:Ne \__keys_show:n
                  {
                    \exp_args:Nc \cs_replacement_spec:N
                    {
                      \c__keys_code_root_str
                      \__keys_trim_spaces:n { #2 / #3 }
                    }
                  }
              }
          }
      }
```

```
      { } { }
  }
\cs_gset:Npx \__keys_show:n #1
  {
    \exp_not:N \__keys_show:w
      #1
      \tl_to_str:n { \__keys_precompile:n }
      #1
      \tl_to_str:n { \__keys_precompile:n }
      \exp_not:N \s__keys_stop
  }
\use:x
  {
    \cs_gset:Npn \exp_not:N \__keys_show:w
      ##1 \tl_to_str:n { \__keys_precompile:n }
      ##2 \tl_to_str:n { \__keys_precompile:n }
      ##3 \exp_not:N \s__keys_stop
  }
  {
    \tl_if_blank:nTF {#2}
      {#1}
      { \__keys_show:Nw #2 \s__keys_stop }
  }
\use:x
  {
    \cs_gset:Npn \exp_not:N \__keys_show:Nw ##1##2
      \c_right_brace_str \exp_not:N \s__keys_stop
  }
  {#2}
```

8.3 Setup

```
⟨@@=prototype⟩
```

\l__prototype_tmp_tl

```
\tl_new:N \l__prototype_tmp_tl
```

(*End definition for* `\l__prototype_tmp_tl`.)

8.4 Data structures

\l__prototype_object_prop

```
\prop_new:N \l__prototype_object_prop
```

(*End definition for* `\l__prototype_object_prop`.)

8.5 Creating objects

\prototype_declare_object:nn
__prototype_declare_object: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.

```
\cs_new_protected:Npn \prototype_declare_object:nn #1#2
  {
    \exp_args:Nx \__prototype_declare_object:nn { \int_eval:n {#2} } {#1}
  }
```

```
\cs_new_protected:Npn \__prototype_declare_object:nn #1#2
  {
    \int_compare:nTF { 0 <= #1 <= 9 }
      {
        \msg_info:nnnn { prototype } { declare-object-type } {#2} {#1}
        \prop_put:Nnn \l__prototype_object_prop {#2} {#1}
      }
      { \msg_error:nnxx { prototype } { bad-number-of-arguments } {#2} {#1} }
  }
```

(*End definition for* `\prototype_declare_object:nn` *and* `\__prototype_declare_object:nn`*. This function is documented on page 3.*)

8.6 Templates and instances

`\l__prototype_assignments_tl`

Used to insert the set keys.

```
\tl_new:N \l__prototype_assignments_tl
```

(*End definition for* `\l__prototype_assignments_tl`.)

`\prototype_declare_template:nnnn`
`\prototype_declare_defaults:nnn`

Creating a template means defining the keys, storing the defaults and creating the function. The defaults are done separately from the other parts as that fits the l3keys pattern but also makes it easy to alter that aspect without changing the core implementation.

```
\cs_new_protected:Npn \prototype_declare_template:nnnn #1#2#3#4
  {
    \prop_get:NnNTF \l__prototype_object_prop {#1} \l__prototype_tmp_tl
      {
        \keys_define:nn { prototype / #1 / #2 } {#3}
        \tl_clear_new:c { l__prototype_defaults_ #1 _ #2 _tl }
        \cs_generate_from_arg_count:cNnn
          { __prototype_template_ #1 _ #2 :w }
          \cs_set_protected:Npn
          { \l__prototype_tmp_tl }
          {
            \tl_use:N \l__prototype_assignments_tl
            #4
          }
      }
      { \msg_error:nnn { prototype } { unknown-object-type } {#1} }
  }
\cs_new_protected:Npn \prototype_declare_defaults:nnn #1#2#3
  {
    \cs_if_exist:cTF { __prototype_template_ #1 _ #2 :w }
      { \tl_set:cn { l__prototype_defaults_ #1 _ #2 _tl } {#3} }
      { \msg_error:nnn { prototype } { unknown-template } {#1} {#2} }
  }
```

(*End definition for* `\prototype_declare_template:nnnn` *and* `\prototype_declare_defaults:nnn`*. These functions are documented on page 4.*)

```
\cs_generate_variant:Nn \keys_precompile:nnN { v , nv }
```

`\prototype_use_template:nnn`
`\prototype_declare_instance:nnnn`
`\__prototype_declare_aux:nnnn`

Using a template and creating an instance are the same thing other than the final step: using the template or storing the key settings. We do not attempt to maximise efficiency

in setting, rather we have a clear approach in which the final assignments may have multiple entries.

```
\cs_new_protected:Npn \prototype_use_template:nnn #1#2#3
  {
    \__prototype_declare_aux:nnnn {#1} {#2} {#3}
      { \use:c { __prototype_template_ #1 _ #2 :w } }
  }
\cs_new_protected:Npn \prototype_declare_instance:nnnn #1#2#3#4
  {
    \__prototype_declare_aux:nnnn {#1} {#2} {#4}
      {
        \tl_clear_new:c { l__prototype_instance_ #1 _ #3 _pars_tl }
        \tl_set_eq:cN { l__prototype_instance_ #1 _ #3 _pars_tl }
          \l__prototype_assignments_tl
         \tl_clear_new:c { l__prototype_instance_ #1 _ #3 _template_tl }
         \tl_set:cn { l__prototype_instance_ #1 _ #3 _template_tl } {#2}
      }
  }
\cs_new_protected:Npn \__prototype_declare_aux:nnnn #1#2#3#4
  {
    \cs_if_exist:cTF { __prototype_template_ #1 _ #2 :w }
      {
        \keys_precompile:nvN
          { prototype / #1 / #2 }
          { l__prototype_defaults_ #1 _ #2 _tl }
          \l__prototype_assignments_tl
        \keys_precompile:nnN { prototype / #1 / #2 } {#3} \l__prototype_tmp_tl
        \tl_put_right:NV \l__prototype_assignments_tl \l__prototype_tmp_tl
        #4
      }
      { \msg_error:nnn { prototype } { unknown-template } {#1} {#2} }
  }
```

(*End definition for* `\prototype_use_template:nnn`, `\prototype_declare_instance:nnnn`, *and* `\__prototype_declare_aux:nnnn`. *These functions are documented on page* ??.)

`\prototype_use_instance:nn`
`\prototype_use_instance:nnn`

Recover the values and insert the code.

```
\cs_new_protected:Npn \prototype_use_instance:nn #1#2
  { \prototype_use_instance:nnn {#1} {#2} { } }
\cs_new_protected:Npn \prototype_use_instance:nnn #1#2#3
  {
    \tl_if_exist:cTF { l__prototype_instance_ #1 _ #2 _template_tl }
      {
        \tl_set_eq:Nc \l__prototype_assignments_tl
          { l__prototype_instance_ #1 _ #2 _pars_tl }
        \tl_if_blank:nF {#3}
          {
            \keys_precompile:vnN
              {
                prototype / #1 /
                \tl_use:c { l__prototype_instance_ #1 _ #2 _template_tl }
              }
              {#3}
              \l__prototype_tmp_tl
```

```
            \tl_put_right:NV \l__prototype_assignments_tl
              \l__prototype_tmp_tl
          }
        \use:c
          {
            __prototype_template_ #1 _
            \tl_use:c { l__prototype_instance_ #1 _ #2 _template_tl }
            :w
          }
      }
      { \msg_error:nnn { prototype } { unknown-instance } {#1} {#2} }
  }
```

(*End definition for* `\prototype_use_instance:nn` *and* `\prototype_use_instance:nnn`*. These functions are documented on page* 5.)

8.7 Showing information

`\prototype_show_template_code:nn`
`\prototype_show_template_defaults:nn`
`\prototype_show_instance_values:nn`

```
\cs_new_protected:Npn \prototype_show_template_code:nn #1#2
  {
    \prop_if_in:NnTF \l__prototype_object_prop {#1}
      { \cs_show:c { __prototype_template_ #1 _ #2 :w } }
      { \msg_error:nnn { prototype } { unknown-object-type } {#1} }
  }
\cs_new_protected:Npn \prototype_show_template_defaults:nn #1#2
  {
    \cs_if_exist:cTF { __prototype_template_ #1 _ #2 :w }
      { \tl_show:c { l__prototype_defaults_ #1 _ #2 _tl } }
      { \msg_error:nnn { prototype } { unknown-template } {#1} {#2} }
  }
\cs_new_protected:Npn \prototype_show_instance_values:nn #1#2
  {
    \tl_if_exist:cTF { l__prototype_instance_ #1 _ #2 _template_tl }
      { \tl_show:c { l__prototype_instance_ #1 _ #2 _pars_tl } }
      { \msg_error:nnn { prototype } { unknown-instance } {#1} {#2} }
  }
```

(*End definition for* `\prototype_show_template_code:nn`*,* `\prototype_show_template_defaults:nn`*, and* `\prototype_show_instance_values:nn`*. These functions are documented on page* 6.)

8.8 Messages

```
\msg_new:nnnn { prototype } { bad-number-of-arguments }
  { Bad~number~of~arguments~for~object~type~'#1'. }
  {
    An~object~may~accept~between~0~and~9~arguments.\\
    You~asked~to~use~#2~arguments:~this~is~not~supported.
  }
\msg_new:nnnn { prototype } { unknown-instance }
  { The~instance~'#2'~of~type~'#1'~is~unknown. }
  {
    You~have~asked~to~use~an~instance~'#2',~
    but~this~has~not~been~created.
```

```
  }
\msg_new:nnnn { prototype } { unknown-object-type }
  { The~object~type~'#1'~is~unknown. }
  { An~object~type~needs~to~be~declared~prior~to~using~it. }
\msg_new:nnnn { prototype } { unknown-template }
  { The~template~'#2'~of~type~'#1'~is~unknown. }
  {
    No~interface~has~been~declared~for~a~template~
    '#2'~of~object~type~'#1'.
  }
\msg_new:nnn { prototype } { declare-object-type }
  { Declaring~object~type~'#1'~taking~#2~argument(s). }
\ExplSyntaxOff
```
⟨/2ekernel⟩