

L^AT_EX 2_ε Interfaces

The L^AT_EX Project*

Released 2026-01-23

Abstract

This document contains user (and programmer) documentation for much of the “new” code added to the L^AT_EX kernel in recent years. Over time, this coverage is expected to be expanded to include the majority of user commands defined in `latex.ltx`.

The individual parts are written at different times and were originally meant to stand on their own. This means that explanations are not necessary in the most logical order, and there is some duplication. Headings are also somewhat inconsistent. We expect this to be sorted out over time, as material is revised to form a consistent whole.

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

Contents

I	Hooks	1
1	L^AT_EX's hook management	2
1.1	Introduction	2
1.2	Package writer interface	2
1.2.1	L ^A T _E X 2 _ε interfaces	2
	Declaring hooks	2
	Special declarations for generic hooks	3
	Using hooks in code	4
	Updating code for hooks	5
	Hook names and default labels	8
	The <code>top-level</code> label	10
	Defining relations between hook code	11
	Querying hooks	13
	Displaying hook code	14
	Debugging hook code	15
1.2.2	L3 programming layer (<code>expl3</code>) interfaces	15
1.2.3	On the order of hook code execution	18
1.2.4	The use of “reversed” hooks	20
1.2.5	Difference between “normal” and “one-time” hooks	21
1.2.6	Generic hooks provided by packages	21
1.2.7	Hooks with arguments	22
1.2.8	Private L ^A T _E X kernel hooks	24
1.2.9	Legacy L ^A T _E X 2 _ε interfaces	24
1.3	L ^A T _E X 2 _ε commands and environments augmented by hooks	25
1.3.1	Generic hooks	25
	Generic hooks for all environments	26
	Generic hooks for commands	27
	Generic hooks provided by file loading operations	27
1.3.2	Hooks provided by <code>\begin{document}</code>	28
1.3.3	Hooks provided by <code>\end{document}</code>	28
1.3.4	Hooks provided by <code>\shipout</code> operations	30
1.3.5	Hooks provided for paragraphs	30
1.3.6	Hooks provided in NFSS commands	30
1.3.7	Hook provided by the mark mechanism	31
2	L^AT_EX's hook management for files	32
2.1	Introduction	32
2.1.1	Provided hooks	32
2.1.2	General hooks for file reading	32
2.1.3	Hooks for package and class files	33
2.1.4	Hooks for <code>\include</code> files	34
2.1.5	High-level interfaces for L ^A T _E X	35
2.1.6	Kernel, class, and package interfaces for L ^A T _E X	36
2.1.7	A sample package for structuring the log output	36

3	Hook management for commands	38
3.1	Introduction	38
3.2	Restrictions and operational details	39
3.2.1	Patching	40
	Timing	40
3.2.2	Command copies	40
3.2.3	Grouping	41
3.2.4	Commands that look ahead	41
3.3	Package author interface	41
3.3.1	Arguments and redefining commands	42
4	Paragraph building and hooks	44
4.1	Introduction	44
4.1.1	The default processing done by the engine	44
4.2	The new mechanism implemented for \LaTeX	46
4.2.1	The provided hooks	47
4.2.2	Altered and newly provided commands	48
4.2.3	Examples	49
	Testing the mechanism	49
	Mark the first paragraph of each <code>itemize</code>	51
4.2.4	Some technical notes	51
	Glue items between paragraphs (found with <code>fancypar</code>)	51
5	The shipout routine: hooks and interfaces	53
5.1	Introduction	53
5.1.1	Overloading the <code>\shipout</code> primitive	53
5.1.2	Provided hooks	55
5.1.3	Legacy \LaTeX commands	56
5.1.4	Special commands for use inside the hooks	57
5.1.5	Provided $\text{Lua}\TeX$ callbacks	57
5.1.6	Information counters	58
5.1.7	Debugging shipout code	58
5.2	Emulating commands from other packages	59
5.2.1	Emulating <code>atbegshi</code>	59
5.2.2	Emulating <code>everyshi</code>	60
5.2.3	Emulating <code>atenddvi</code>	60
5.2.4	Emulating <code>everypage</code>	60
II	Run data and page design	61

6	The marks mechanism	62
6.1	Introduction	62
6.2	Design-level and code-level interfaces	63
6.2.1	Use cases for conditionals	65
6.2.2	Understanding regions	65
6.2.3	Debugging mark code	67
6.3	Application examples	67
6.4	Legacy L ^A T _E X 2 _ε interface	67
6.4.1	Legacy design-level and document-level interfaces	68
6.4.2	Legacy interface extensions	68
6.5	Notes on the mechanism	69
6.6	Public interfaces for packages such as <code>multicol</code>	70
6.7	Internal functions for the standard output routine of L ^A T _E X	71
7	Recording and cross-referencing document properties	72
7.1	Introduction	72
7.2	Design discussion	72
7.3	Handling unknown labels and properties	73
7.4	Rerun messages	73
7.5	Open points	73
7.6	Code interfaces	74
7.7	Auxiliary file interfaces	75
7.8	L ^A T _E X 2 _ε interface	76
7.9	Pre-declared properties	77
III	Design-level tools	78
8	L^AT_EX's socket management	79
8.1	Introduction	79
8.2	Configuration of the transformation process	79
8.2.1	The template mechanism	79
8.2.2	The hook mechanism	80
8.2.3	The socket mechanism	81
	Examples	82
	Details and semantics	83
	Command syntax	85
	Rationale for error handling	87

9	Templates: Prototype document functions	88
9.1	Introduction	88
9.2	What is a document?	89
9.3	Types, templates, and instances	89
9.4	Template types	89
9.5	Templates	90
9.6	Multiple choices	94
9.7	Instances	95
9.8	Document interface	96
9.9	Changing existing definitions	96
9.9.1	Expanding the values of keys	97
9.10	Getting information about templates and instances	97
	Index	99

Part I
Hooks

Chapter 1

L^AT_EX's hook management

1.1 Introduction

Hooks are points in the code of commands or environments where it is possible to add processing code into existing commands. This can be done by different packages that do not know about each other, and to allow for hopefully safe processing it is necessary to sort different chunks of code added by different packages into a suitable processing order.

This is done by the packages adding chunks of code (via `\AddToHook`) and labeling their code with some label by default using the package name as a label.

At `\begin{document}` all code for a hook is then sorted according to some rules (given by `\DeclareHookRule`) for fast execution without processing overhead. If the hook code is modified afterwards (or the rules are changed), a new version for fast processing is generated.

Some hooks are used already in the preamble of the document. If that happens then the hook is prepared for execution (and sorted) already at that point.

1.2 Package writer interface

The hook management system is offered as a set of CamelCase commands for traditional L^AT_EX 2_ε packages (and for use in the document preamble if needed) as well as `expl3` commands for modern packages, that use the L3 programming layer of L^AT_EX. Behind the scenes, a single set of data structures is accessed so that packages from both worlds can coexist and access hooks in other packages.

1.2.1 L^AT_EX 2_ε interfaces

Declaring hooks

With a few exceptions, hooks have to be declared before they can be used. The exceptions are the generic hooks for commands and environments (executed at `\begin` and `\end`), and the generic hooks run when loading files (see section [1.3.1](#)).

`\NewHook` `\NewHook {<hook>}`

Creates a new `<hook>`. If this hook is declared within a package it is suggested that its name is always structured as follows: `<package-name>/<hook-name>`. If necessary you can further subdivide the name by adding more `/` parts. If a hook name is already taken, an error is raised and the hook is not created.

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section 1.2.1. The string `??` can't be used as a hook name because it has a special significance as a placeholder in hook rules.

`\NewReversedHook` `\NewReversedHook {<hook>}`

Like `\NewHook` declares a new `<hook>`, the difference is that the code chunks for this hook are in reverse order by default (those added last are executed first). Any rules for the hook are applied after the default ordering. See sections 1.2.3 and 1.2.4 for further details.

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

`\NewMirroredHookPair` `\NewMirroredHookPair {<hook-1>} {<hook-2>}`

A shorthand for `\NewHook{<hook-1>}\NewReversedHook{<hook-2>}`.

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

`\NewHookWithArguments` `\NewHookWithArguments {<hook>} {<number>}`

Creates a new `<hook>` whose code takes `<number>` arguments, and otherwise works exactly like `\NewHook`. Section 1.2.7 explains hooks with arguments.

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

`\NewReversedHookWithArguments` `\NewReversedHookWithArguments {<hook>} {<number>}`

Like `\NewReversedHook`, but creates a hook whose code takes `<number>` arguments. Section 1.2.7 explains hooks with arguments.

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

`\NewMirroredHookPairWithArguments` `\NewMirroredHookPairWithArguments {<hook-1>} {<hook-2>} {<number>}`

A shorthand for `\NewHookWithArguments{<hook-1>}{<number>}`

`\NewReversedHookWithArguments{<hook-2>}{<number>}`. Section 1.2.7 explains hooks with arguments.

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

Special declarations for generic hooks

The declarations here should normally not be used. They are available to provide support for special use cases mainly involving generic command hooks.

`\DisableGenericHook` `\DisableGenericHook` $\langle hook \rangle$

After this declaration¹ the $\langle hook \rangle$ is no longer usable: Any further attempt to add code to it will result in an error and any use, e.g., via `\UseHook`, will simply do nothing.

This is intended to be used with generic command hooks (see `ltxcmdhooks-doc`) as depending on the definition of the command such generic hooks may be unusable. If that is known, a package developer can disable such hooks up front.

The $\langle hook \rangle$ can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

`\ActivateGenericHook` `\ActivateGenericHook` $\langle hook \rangle$

This declaration activates a generic hook provided by a package/class (e.g., one used in code with `\UseHook` or `\UseOneTimeHook`) without it being explicitly declared with `\NewHook`). If the hook is already activated, this command does nothing.

Note that this command does not undo the effect of `\DisableGenericHook`. See section 1.2.6 for a discussion of when this declaration is appropriate.

Using hooks in code

Using a hook that is executing the code that has been associated with it is only allowed if the hook has been previously declared with `\NewHook`. For performance reason there are no runtime checks for this and it is the responsibility of the programmer of a package to ensure that all hooks that are used in a package (with one of the commands in this section) are declared first.

`\UseHook` `\UseHook` $\langle hook \rangle$

Execute the code stored in the $\langle hook \rangle$.

Before `\begin{document}` the fast execution code for a hook is not set up, so in order to use a hook there it is explicitly initialized first. As that involves assignments using a hook at those times is not 100% the same as using it after `\begin{document}`.

The $\langle hook \rangle$ *cannot* be specified using the dot-syntax. A leading `.` is treated literally.

`\UseHookWithArguments` `\UseHookWithArguments` $\langle hook \rangle$ $\langle number \rangle$ $\langle arg_1 \rangle$... $\langle arg_n \rangle$

Execute the code stored in the $\langle hook \rangle$ and pass the arguments $\langle arg_1 \rangle$ through $\langle arg_n \rangle$ to the $\langle hook \rangle$. Otherwise, it works exactly like `\UseHook`. The $\langle number \rangle$ should be the number of arguments declared for the hook. If the hook is not declared, this command does nothing and it will remove $\langle number \rangle$ items from the input. Section 1.2.7 explains hooks with arguments.

The $\langle hook \rangle$ *cannot* be specified using the dot-syntax. A leading `.` is treated literally.

¹In the 2020/06 release this command was called `\DisableHook`, but that name was misleading as it shouldn't be used to disable non-generic hooks.

`\UseOneTimeHook` `\UseOneTimeHook` $\langle hook \rangle$

Some hooks are only used (and can be only used) in one place, for example, those in `\begin{document}` or `\end{document}`. From that point onwards, adding to the hook through a defined $\langle addto-cmd \rangle$ command (e.g., `\AddToHook` or `\AtBeginDocument`, etc.) would have no effect (as would the use of such a command inside the hook code itself). It is therefore customary to redefine $\langle addto-cmd \rangle$ to simply process its argument, i.e., essentially make it behave like `\@firstofone`.

`\UseOneTimeHook` does that: it records that the hook has been consumed and any further attempt to add to it will result in executing the code to be added immediately.

Using `\UseOneTimeHook` several times with the same $\langle hook \rangle$ means that it only executes the first time it is used. For example, if it is used in a command that can be called several times then the hook executes during only the *first* invocation of that command; this allows its use as an “initialization hook”.

Mixing `\UseHook` and `\UseOneTimeHook` for the same $\langle hook \rangle$ should be avoided, but if this is done then neither will execute after the first `\UseOneTimeHook`.

The $\langle hook \rangle$ *cannot* be specified using the dot-syntax. A leading `.` is treated literally. See section 1.2.1 for details.

`\UseOneTimeHookWithArguments` `\UseOneTimeHookWithArguments` $\langle hook \rangle$ $\langle number \rangle$ $\langle arg_1 \rangle$... $\langle arg_n \rangle$

Works exactly like `\UseOneTimeHook`, but passes arguments $\langle arg_1 \rangle$ through $\langle arg_n \rangle$ to the $\langle hook \rangle$. The $\langle number \rangle$ should be the number of arguments declared for the hook. If the hook is not declared, this command does nothing and it will remove $\langle number \rangle$ items from the input.

It should be noted that after a one-time hook is used, it is no longer possible to use `\AddToHookWithArguments` or similar with that hook. `\AddToHook` continues to work as normal. Section 1.2.7 explains hooks with arguments.

The $\langle hook \rangle$ *cannot* be specified using the dot-syntax. A leading `.` is treated literally. See section 1.2.1 for details.

Updating code for hooks

In contrast to the commands from the previous section, declarations such as `\AddToHook` or `\DeclareHookRule` can be used even when the hook is not yet declared. The rationale is that the hook declaration may be in some package that is loaded later, or perhaps not loaded at all.

A side effect of this design is that misspellings do not raise an error but are simply regarded as declarations for hooks with a different name.

`\AddToHook` `\AddToHook <hook> [<label>] <code>`

Adds `<code>` to the `<hook>` labeled by `<label>`. When the optional argument `<label>` is not provided, the `<default label>` is used (see section 1.2.1). If `\AddToHook` is used in a package/class, the `<default label>` is the package/class name, otherwise it is `top-level` (the `top-level` label is treated differently: see section 1.2.1).

If there already exists code under the `<label>` then the new `<code>` is appended to the existing one (even if this is a reversed hook). If you want to replace existing code under the `<label>`, first apply `\RemoveFromHook`.

The hook doesn't have to exist for code to be added to it. However, if it is not declared, then obviously the added `<code>` will never be executed. This allows for hooks to work regardless of package loading order and enables packages to add to hooks from other packages without worrying whether they are actually used in the current document. See section 1.2.1.

The `<hook>` and `<label>` can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

`\AddToHookWithArguments` `\AddToHookWithArguments <hook> [<label>] <code>`

Works exactly like `\AddToHook`, except that the `<code>` can access the arguments passed to the hook using `#1`, `#2`, ..., `#n` (up to the number of arguments declared for the hook). If the `<code>` should contain *parameter tokens* (`#`) that are not supposed to be understood as the arguments of the hook, such tokens should be doubled. For example, with `\AddToHook` one can write:

```
\AddToHook{myhook}{\def\foo#1{Hello, #1!}}
```

but to achieve the same with `\AddToHookWithArguments`, one should write:

```
\AddToHookWithArguments{myhook}{\def\foo##1{Hello, ##1!}}
```

because in the latter case, `#1` refers to the first argument of the hook `myhook`. Section 1.2.7 explains hooks with arguments.

The `<hook>` and `<label>` can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

`\RemoveFromHook` `\RemoveFromHook <hook> [<label>]`

Removes any code labeled by `<label>` from the `<hook>`. When the optional argument `<label>` is not provided, the `<default label>` is used (see section 1.2.1).

If there is no code under the `<label>` in the `<hook>`, or if the `<hook>` does not exist, a warning is issued when you attempt to `\RemoveFromHook`, and the command is ignored. `\RemoveFromHook` should be used only when you know exactly what labels are in a hook.

Typically this will be when some code gets added to a hook by a package, then later this code is removed by that same package. If you want to prevent the execution of code from another package, use the `voids` rule instead (see section 1.2.1).

If the optional `<label>` argument is `*`, then all code chunks are removed. This is rather dangerous as it may well drop code from other packages (that one may not know about); it should therefore not be used in packages but only in document preambles!

The `<hook>` and `<label>` can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

Important:
The `\RemoveFromHook` command should be only used if one has full control over the code chunk to be removed. In particular it should not be used to remove code chunks from other packages! For this the `voids` relation is provided.

In contrast to the `voids` relationship between two labels in a `\DeclareHookRule` this is a destructive operation as the labeled code is removed from the hook data structure, whereas the relationship setting can be undone by providing a different relationship later.

A useful application for this declaration inside the document body is when one wants to temporarily add code to hooks and later remove it again, e.g.,

```
\AddToHook{env/quote/begin}{\small}
\begin{quote}
  A quote set in a smaller typeface
\end{quote}
...
\RemoveFromHook{env/quote/begin}
... now back to normal for further quotes
```

Note that you can't cancel the setting with

```
\AddToHook{env/quote/begin}{}
```

because that only “adds” a further empty chunk of code to the hook. Adding `\normalsize` would work but that means the hook then contained `\small\normalsize` which means two font size changes for no good reason.

The above is only needed if one wants to typeset several quotes in a smaller typeface. If the hook is only needed once then `\AddToHookNext` is simpler, because it resets itself after one use.

`\AddToHookNext` `\AddToHookNext {<hook>} {<code>}`

Adds `<code>` to the next invocation of the `<hook>`. The code is executed after the normal hook code has finished and it is executed only once, i.e. it is deleted after it was used.

Using this declaration is a global operation, i.e., the code is not lost even if the declaration is used inside a group and the next invocation of the hook happens after the end of that group. If the declaration is used several times before the hook is executed then all code is executed in the order in which it was declared.²

If this declaration is used with a one-time hook then the code is only ever used if the declaration comes before the hook's invocation. This is because, in contrast to `\AddToHook`, the code in this declaration is not executed immediately in the case when the invocation of the hook has already happened—in other words, this code will truly execute only on the next invocation of the hook (and in the case of a one-time hook there is no such “next invocation”). This gives you a choice: should my code execute always, or should it execute only at the point where the one-time hook is used (and not at all if this is impossible)? For both of these possibilities there are use cases.

It is possible to nest this declaration using the same hook (or different hooks): e.g.,

```
\AddToHookNext{<hook>}{<code-1>\AddToHookNext{<hook>}{<code-2>}}
```

will execute `<code-1>` next time the `<hook>` is used and at that point puts `<code-2>` into the `<hook>` so that it gets executed on following time the hook is run.

A hook doesn't have to exist for code to be added to it. This allows for hooks to work regardless of package loading order. See section 1.2.1.

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

²There is no mechanism to reorder such code chunks (or delete them).

`\AddToHookNextWithArguments` `\AddToHookNextWithArguments {<hook>} {<code>}`

Works exactly like `\AddToHookNext`, but the `<code>` can contain references to the arguments of the `<hook>` as described for `\AddToHookWithArguments` above. Section 1.2.7 explains hooks with arguments.

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

`\ClearHookNext` `\ClearHookNext {<hook>}`

Normally `\AddToHookNext` is only used when you know precisely where it will apply and why you want some extra code at that point. However, there are a few use cases in which such a declaration needs to be canceled, for example, when discarding a page with `\DiscardShipoutBox` (but even then not always), and in such situations `\ClearHookNext` can be used.

Hook names and default labels

It is best practice to use `\AddToHook` in packages or classes *without specifying a `<label>`* because then the package or class name is automatically used, which is helpful if rules are needed, and avoids mistyping the `<label>`.

Using an explicit `<label>` is only necessary in very specific situations, e.g., if you want to add several chunks of code into a single hook and have them placed in different parts of the hook (by providing some rules).

The other case is when you develop a larger package with several sub-packages. In that case you may want to use the same `<label>` throughout the sub-packages in order to avoid that the labels change if you internally reorganize your code.

Except for `\UseHook`, `\UseOneTimeHook` and `\IfHookEmptyTF` (and their `expl3` interfaces `\hook_use:n`, `\hook_use_once:n` and `\hook_if_empty:nTF`), all `<hook>` and `<label>` arguments are processed in the same way: first, spaces are trimmed around the argument, then it is fully expanded until only character tokens remain. If the full expansion of the `<hook>` or `<label>` contains a non-expandable non-character token, a low-level `TeX` error is raised (namely, the `<hook>` is expanded using `TeX`'s `\csname... \endcsname`, as such, Unicode characters are allowed in `<hook>` and `<label>` arguments). The arguments of `\UseHook`, `\UseOneTimeHook`, and `\IfHookEmptyTF` are processed much in the same way except that spaces are not trimmed around the argument, for better performance.

It is not enforced, but highly recommended that the hooks defined by a package, and the `<labels>` used to add code to other hooks contain the package name to easily identify the source of the code chunk and to prevent clashes. This should be the standard practice, so this hook management code provides a shortcut to refer to the current package in the name of a `<hook>` and in a `<label>`. If the `<hook>` name or the `<label>` consist just of a single dot (`.`), or starts with a dot followed by a slash (`./`) then the dot denotes the `<default label>` (usually the current package or class name—see `\SetDefaultHookLabel`). A “.” or “./” anywhere else in a `<hook>` or in `<label>` is treated literally and is not replaced.

For example, inside the package `mypackage.sty`, the default label is `mypackage`, so the instructions:

```
\NewHook {./hook}
\AddToHook {./hook}[.]{code} % Same as \AddToHook{./hook}{code}
```

```

\AddToHook {./hook}[./sub]{code}
\DeclareHookRule{begindocument}{.}{before}{babel}
\AddToHook {file/foo.tex/after}{code}

```

are equivalent to:

```

\NewHook {mypackage/hook}
\AddToHook {mypackage/hook}[mypackage]{code}
\AddToHook {mypackage/hook}[mypackage/sub]{code}
\DeclareHookRule{begindocument}{mypackage}{before}{babel}
\AddToHook {file/foo.tex/after}{code} % unchanged

```

The `<default label>` is automatically set equal to the name of the current package or class at the time the package is loaded. If the hook command is used outside of a package, or the current file wasn't loaded with `\usepackage` or `\documentclass`, then the `top-level` is used as the `<default label>`. This may have exceptions—see `\PushDefaultHookLabel`.

This syntax is available in all `<label>` arguments and most `<hook>` arguments, both in the $\text{\LaTeX} 2_{\epsilon}$ interface, and the $\text{\LaTeX} 3$ interface described in section 1.2.2.

Note, however, that the replacement of `.` by the `<default label>` takes place when the hook command is executed, so actions that are somehow executed after the package ends will have the wrong `<default label>` if the dot-syntax is used. For that reason, this syntax is not available in `\UseHook` (and `\hook_use:n`) because the hook is most of the time used outside of the package file in which it was defined. This syntax is also not available in the hook conditionals `\IfHookEmptyTF` (and `\hook_if_empty:nTF`), because these conditionals are used in some performance-critical parts of the hook management code, and because they are usually used to refer to other package's hooks, so the dot-syntax doesn't make much sense.

In some cases, for example in large packages, one may want to separate the code in logical parts, but still use the main package name as the `<label>`, then the `<default label>` can be set using `\PushDefaultHookLabel{...}\PopDefaultHookLabel` or `\SetDefaultHookLabel{...}`.

Important:
The dot-syntax is not available with `\UseHook` and some other commands that are typically used within code!

<code>\PushDefaultHookLabel</code>	<code>\PushDefaultHookLabel {⟨<i>default label</i>⟩}</code>
<code>\PopDefaultHookLabel</code>	<code>⟨<i>code</i>⟩</code>
	<code>\PopDefaultHookLabel</code>

`\PushDefaultHookLabel` sets the current `⟨default label⟩` to be used in `⟨label⟩` arguments, or when replacing a leading “.” (see above). `\PopDefaultHookLabel` reverts the `⟨default label⟩` to its previous value.

Inside a package or class, the `⟨default label⟩` is equal to the package or class name, unless explicitly changed. Everywhere else, the `⟨default label⟩` is `top-level` (see section 1.2.1) unless explicitly changed.

The effect of `\PushDefaultHookLabel` holds until the next `\PopDefaultHookLabel`. `\usepackage` (and `\RequirePackage` and `\documentclass`) internally use

```
\PushDefaultHookLabel{⟨package name⟩}
  ⟨package code⟩
\PopDefaultHookLabel
```

to set the `⟨default label⟩` for the package or class file. Inside the `⟨package code⟩` the `⟨default label⟩` can also be changed with `\SetDefaultHookLabel`. `\input` and other file input-related commands from the L^AT_EX kernel do not use `\PushDefaultHookLabel`, so code within files loaded by these commands does *not* get a dedicated `⟨label⟩`! (that is, the `⟨default label⟩` is the current active one when the file was loaded.)

Packages that provide their own package-like interfaces (TikZ’s `\usetikzlibrary`, for example) can use `\PushDefaultHookLabel` and `\PopDefaultHookLabel` to set dedicated labels and to emulate `\usepackage`-like hook behavior within those contexts.

The `top-level` label is treated differently, and is reserved to the user document, so it is not allowed to change the `⟨default label⟩` to `top-level`.

<code>\SetDefaultHookLabel</code>	<code>\SetDefaultHookLabel {⟨<i>default label</i>⟩}</code>
-----------------------------------	--

Similarly to `\PushDefaultHookLabel`, sets the current `⟨default label⟩` to be used in `⟨label⟩` arguments, or when replacing a leading “.”. The effect holds until the label is changed again or until the next `\PopDefaultHookLabel`. The difference between `\PushDefaultHookLabel` and `\SetDefaultHookLabel` is that the latter does not save the current `⟨default label⟩`.

This command is useful when a large package is composed of several smaller packages, but all should have the same `⟨label⟩`, so `\SetDefaultHookLabel` can be used at the beginning of each package file to set the correct label.

`\SetDefaultHookLabel` is not allowed in the main document, where the `⟨default label⟩` is `top-level` and there is no `\PopDefaultHookLabel` to end its effect. It is also not allowed to change the `⟨default label⟩` to `top-level`.

The top-level label

The `top-level` label, assigned to code added from the main document, is different from other labels. Code added to hooks (usually `\AtBeginDocument`) in the preamble is almost always to change something defined by a package, so it should go at the very end of the hook.

Therefore, code added in the `top-level` is always executed at the end of the hook, regardless of where it was declared. If the hook is reversed (see `\NewReversedHook`), the `top-level` chunk is executed at the very beginning instead.

Rules regarding `top-level` have no effect: if a user wants to have a specific set of rules for a code chunk, they should use a different label to said code chunk, and provide a rule for that label instead.

The `top-level` label is exclusive for the user, so trying to add code with that label from a package results in an error.

Defining relations between hook code

The default assumption is that code added to hooks by different packages are independent and the order in which they are executed is irrelevant. While this is true in many cases it is obviously false in others.

Before the hook management system was introduced packages had to take elaborate precautions to determine whether some other package had also been loaded (before or after) and then to find some ways to alter its behavior accordingly. In addition it was often the user's responsibility to load packages in the right order so that alterations made by packages were done in that same order; and in some cases even altering the loading order wouldn't resolve the conflicts.

With the new hook management system it is now possible to define rules (i.e., relationships) between code chunks added by different packages and to specify explicitly the order in which they should be processed.

The rules can be declared for hooks before the hook has been declared with `\NewHook` and they are allowed to refer to code labels that do not yet exist, e.g., because a package defining the code chunk with that label has not yet been loaded. When the hook code is finally sorted for fast execution, all rules that apply are acted on and the others are ignored.

This offers the flexibility needed to handle complicated relationships between code from different packages and to set this up beforehand in a way that is independent of whether or not the packages are actually loaded in a specific document. The downside of this is that misspellings of hook names or code labels will not raise any error, instead the rule will simply never apply!

`\DeclareHookRule` `\DeclareHookRule` $\langle hook \rangle$ $\langle label1 \rangle$ $\langle relation \rangle$ $\langle label2 \rangle$

Defines a relation between $\langle label1 \rangle$ and $\langle label2 \rangle$ for a given $\langle hook \rangle$. If $\langle hook \rangle$ is ?? this defines a default relation for all hooks that use the two labels, i.e., that have chunks of code labeled with $\langle label1 \rangle$ and $\langle label2 \rangle$.

Currently, the supported relations are the following:

`before` or `<` Code for $\langle label1 \rangle$ comes before code for $\langle label2 \rangle$.

`after` or `>` Code for $\langle label1 \rangle$ comes after code for $\langle label2 \rangle$.

`incompatible-warning` Only code for either $\langle label1 \rangle$ or $\langle label2 \rangle$ can appear for that hook (a way to say that two packages—or parts of them—are incompatible). A warning is raised if both labels appear in the same hook.

`incompatible-error` Like `incompatible-error` but instead of a warning a L^AT_EX error is raised, and the code for both labels are dropped from that hook until the conflict is resolved.

`voids` Code for $\langle label1 \rangle$ overwrites code for $\langle label2 \rangle$. More precisely, code for $\langle label2 \rangle$ is dropped for that hook. This can be used, for example if one package is a superset in functionality of another one and therefore wants to undo code in some hook and replace it with its own version.

`unrelated` The order of code for $\langle label1 \rangle$ and $\langle label2 \rangle$ is irrelevant. This rule is there to undo an incorrect rule specified earlier.

There can only be a single relation between two labels for a given hook, i.e., a later `\DeclareHookRule` overwrites any previous declaration. In all cases rules specific to a given hook take precedence over default rules that use ?? as the $\langle hook \rangle$.

If a default rule is applied, it is done before reversing the label order in a reversed hook, e.g., `before` in a default rule effectively becomes `after` in such a hook. In contrast, a rule for a specific hook is always applied to the state after any reversal (i.e., the state you see when using `\ShowHook` on that hook).

The $\langle hook \rangle$ and $\langle label \rangle$ can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

`\ClearHookRule` `\ClearHookRule` $\langle hook \rangle$ $\langle label1 \rangle$ $\langle label2 \rangle$

Syntactic sugar for saying that $\langle label1 \rangle$ and $\langle label2 \rangle$ are unrelated for the given $\langle hook \rangle$.

`\DeclareDefaultHookRule` `\DeclareDefaultHookRule` $\langle label1 \rangle$ $\langle relation \rangle$ $\langle label2 \rangle$

This sets up a relation between $\langle label1 \rangle$ and $\langle label2 \rangle$ for all hooks unless overwritten by a specific rule for a hook. Useful for cases where one package has a specific relation to some other package, e.g., is `incompatible` or always needs a special ordering `before` or `after`. (Technically it is just a shorthand for using `\DeclareHookRule` with `??` as the hook name.)

If such a rule is applied to a reversed hook it behaves as if the rule is reversed (e.g., `after` becomes `before`) because those rules are applied first and then the order is reversed. The rationale is that in hook pairs (in which the ordering in one is reversed) default rules have to be reversed too in nearly all scenarios. If this is not the case, a default rule can't be used or has to be overwritten with an explicit `\DeclareHookRule` for that specific hook.

Declaring default rules is only supported in the document preamble.³

The $\langle label \rangle$ can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

Querying hooks

Simpler data types, like token lists, have three possible states; they can:

- exist and be empty;
- exist and be non-empty; and
- not exist (in which case emptiness doesn't apply);

Hooks are a bit more complicated: a hook may exist or not, and independently it may or may not be empty. This means that even a hook that doesn't exist may be non-empty and it can also be disabled.

This seemingly strange state may happen when, for example, package *A* defines hook `A/foo`, and package *B* adds some code to that hook. However, a document may load package *B* before package *A*, or may not load package *A* at all. In both cases some code is added to hook `A/foo` without that hook being defined yet, thus that hook is said to be non-empty, whereas it doesn't exist. Therefore, querying the existence of a hook doesn't imply its emptiness, neither does the other way around.

Given that code or rules can be added to a hook even if it doesn't physically exist yet, means that a querying its existence has no real use case (in contrast to other variables that can only be update if they have already been declared). For that reason only the test for emptiness has a public interface.

A hook is said to be empty when no code was added to it, either to its permanent code pool, or to its "next" token list. The hook doesn't need to be declared to have code added to its code pool. A hook is said to exist when it was declared with `\NewHook` or some variant thereof. Generic hooks such as `file` and `env` hooks are automatically declared when code is added to them.

³Trying to do so, e.g., via `\DeclareHookRule` with `??` has bad side-effects and is not supported (though not explicitly caught for performance reasons).

<code>\IfHookEmptyTF</code>	*	<code>\IfHookEmptyTF {<hook>} {<true code>} {<false code>}</code>
<code>\IfHookEmptyT</code>	*	Tests if the <code><hook></code> is empty (<i>i.e.</i> , no code was added to it using either <code>\AddToHook</code> or <code>\AddToHookNext</code>) or such code was removed again (via <code>\RemoveFromHook</code>), and branches to either <code><true code></code> or <code><false code></code> depending on the result.
<code>\IfHookEmptyF</code>	*	

The `<hook>` *cannot* be specified using the dot-syntax. A leading `.` is treated literally.

Displaying hook code

If one has to adjust the code execution in a hook using a hook rule it is helpful to get some information about the code associated with a hook, its current order and the existing rules.

<code>\ShowHook</code>	<code>\ShowHook {<hook>}</code>
<code>\LogHook</code>	<code>\LogHook {<hook>}</code>

Displays information about the `<hook>` such as

- the code chunks (and their labels) added to it,
- any rules set up to order them,
- the computed order in which the chunks are executed,
- any code executed on the next invocation only.

`\LogHook` prints the information to the `.log` file, and `\ShowHook` prints them to the terminal/command window and starts TeX's prompt (only in `\errorstopmode`) to wait for user action.

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section [1.2.1](#).

Suppose a hook `example-hook` whose output of `\ShowHook{example-hook}` is:

```

1  -> The hook 'example-hook':
2  > Code chunks:
3  >   foo -> [code from package 'foo']
4  >   bar -> [from package 'bar']
5  >   baz -> [package 'baz' is here]
6  > Document-level (top-level) code (executed last):
7  >   -> [code from 'top-level']
8  > Extra code for next invocation:
9  >   -> [one-time code]
10 > Rules:
11 >   foo|baz with relation >
12 >   baz|bar with default relation <
13 > Execution order (after applying rules):
14 >   baz, foo, bar.
```

In the listing above, lines 3 to 5 show the three code chunks added to the hook and their respective labels in the format

`<label> -> <code>`

Line 7 shows the code chunk added by the user in the main document (labeled `top-level`) in the format

```
Document-level (top-level) code (executed  $\langle first/last \rangle$ ):
  ->  $\langle top-level code \rangle$ 
```

This code will be either the first or last code executed by the hook (`last` if the hook is normal, `first` if it is reversed). This chunk is not affected by rules and does not take part in sorting.

Line 9 shows the code chunk for the next execution of the hook in the format

```
->  $\langle next-code \rangle$ 
```

This code will be used and disappear at the next `\UseHook{example-hook}`, in contrast to the chunks mentioned earlier, which can only be removed from that hook by doing `\RemoveFromHook{label}[example-hook]`.

Lines 11 and 12 show the rules declared that affect this hook in the format

```
 $\langle label-1 \rangle | \langle label-2 \rangle$  with  $\langle default? \rangle$  relation  $\langle relation \rangle$ 
```

which means that the $\langle relation \rangle$ applies to $\langle label-1 \rangle$ and $\langle label-2 \rangle$, in that order, as detailed in `\DeclareHookRule`. If the relation is `default` it means that this rule applies to $\langle label-1 \rangle$ and $\langle label-2 \rangle$ in *all* hooks, (unless overridden by a non-default relation).

Finally, line 14 lists the labels in the hook after sorting; that is, in the order they will be executed when the hook is used.

Debugging hook code

```
\DebugHooksOn \DebugHooksOn ... \DebugHooksOff
\DebugHooksOff
```

Turn the debugging of hook code on or off. This displays most changes made to the hook data structures. The output is rather coarse and not really intended for normal use, but it can be helpful in case hooks do not work as expected. See also [1.2.1](#) for commands to inspect individual hooks.

1.2.2 L3 programming layer (expl3) interfaces

This is a quick summary of the L^AT_EX₃ programming interfaces for use with packages written in `expl3`. In contrast to the L^AT_EX_{2 ϵ} interfaces they always use mandatory arguments only, e.g., you always have to specify the $\langle label \rangle$ for a code chunk. We therefore suggest to use the declarations discussed in the previous section even in `expl3` packages, but the choice is yours.

```
\hook_new:n \hook_new:n {hook}
\hook_new_reversed:n \hook_new_reversed:n {hook}
\hook_new_pair:nn \hook_new_pair:nn {hook-1} {hook-2}
```

Creates a new $\langle hook \rangle$ with normal or reverse ordering of code chunks. `\hook_new_pair:nn` creates a pair of such hooks with $\{hook-2\}$ being a reversed hook. If a hook name is already taken, an error is raised and the hook is not created.

The $\langle hook \rangle$ can be specified using the dot-syntax to denote the current package name. See section [1.2.1](#).

<code>\hook_new_with_args:nn</code>	<code>\hook_new_with_args:nn {<hook>} {<number>}</code>
<code>\hook_new_reversed_with_args:nn</code>	<code>\hook_new_reversed_with_args:nn {<hook>} {<number>}</code>
<code>\hook_new_pair_with_args:nmn</code>	<code>\hook_new_pair_with_args:nmn {<hook-1>} {<hook-2>} {<number>}</code>

Creates a new `<hook>` with normal or reverse ordering of code chunks, that takes `<number>` arguments from the input stream when it is used. `\hook_new_pair_with_args:nn` creates a pair of such hooks with `{<hook-2>}` being a reversed hook. If a hook name is already taken, an error is raised and the hook is not created.

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

<code>\hook_disable_generic:n</code>	<code>\hook_disable_generic:n {<hook>}</code>
--------------------------------------	---

Marks `{<hook>}` as disabled. Any further attempt to add code to it or declare it, will result in an error and any call to `\hook_use:n` will simply do nothing.

This declaration is intended for use with generic hooks that are known not to work (see `ltxcmdhooks-doc`) if they receive code.

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

<code>\hook_activate_generic:n</code>	<code>\hook_activate_generic:n {<hook>}</code>
---------------------------------------	--

This is like `\hook_new:n` but it does nothing if the hook was previously declared with `\hook_new:n`. This declaration should be used only in special situations, e.g., when a command from another package needs to be altered and it is not clear whether a generic `cmd` hook (for that command) has been previously explicitly declared.

Normally `\hook_new:n` should be used instead of this.

<code>\hook_use:n</code>	<code>\hook_use:n {<hook>}</code>
<code>\hook_use:nmw</code>	<code>\hook_use:nmw {<hook>} {<number>} {<arg₁>} ... {<arg_n>}</code>

Executes the `{<hook>}` code followed (if set up) by the code for next invocation only, then empties that next invocation code. `\hook_use:nmw` should be used for hooks declared with arguments, and should be followed by as many brace groups as the declared number of arguments. The `<number>` should be the number of arguments declared for the hook. If the hook is not declared, this command does nothing and it will remove `<number>` items from the input.

The `<hook>` *cannot* be specified using the dot-syntax. A leading `.` is treated literally.

<code>\hook_use_once:n</code>	<code>\hook_use_once:n {<hook>}</code>
<code>\hook_use_once:nmw</code>	<code>\hook_use_once:nmw {<hook>} {<number>} {<arg₁>} ... {<arg_n>}</code>

Changes the `{<hook>}` status so that from now on any addition to the hook code is executed immediately. Then execute any `{<hook>}` code already set up. `\hook_use_once:nmw` should be used for hooks declared with arguments, and should be followed by as many brace groups as the declared number of arguments. The `<number>` should be the number of arguments declared for the hook. If the hook is not declared, this command does nothing and it will remove `<number>` items from the input.

The `<hook>` *cannot* be specified using the dot-syntax. A leading `.` is treated literally.

<code>\hook_gput_code:nnn</code>	<code>\hook_gput_code:nnn</code>	<code>{\hook} {\label} {\code}</code>
<code>\hook_gput_code_with_args:nnn</code>	<code>\hook_gput_code_with_args:nnn</code>	<code>{\hook} {\label} {\code}</code>

Adds a chunk of `<code>` to the `<hook>` labeled `<label>`. If the label already exists the `<code>` is appended to the already existing code.

If `\hook_gput_code_with_args:nnn` is used, the `<code>` can access the arguments passed to `\hook_use:nnw` (or `\hook_use_once:nnw`) with `#1`, `#2`, ..., `#n` (up to the number of arguments declared for the hook). In that case, if an actual parameter token should be added to the code, it should be doubled.

If code is added to an external `<hook>` (of the kernel or another package) then the convention is to use the package name as the `<label>` not some internal module name or some other arbitrary string.

The `<hook>` and `<label>` can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

<code>\hook_gput_next_code:nn</code>	<code>\hook_gput_next_code:nn</code>	<code>{\hook} {\code}</code>
<code>\hook_gput_next_code_with_args:nn</code>	<code>\hook_gput_next_code_with_args:nn</code>	<code>{\hook} {\code}</code>

Adds a chunk of `<code>` for use only in the next invocation of the `<hook>`. Once used it is gone.

If `\hook_gput_next_code_with_args:nn` is used, the `<code>` can access the arguments passed to `\hook_use:nnw` (or `\hook_use_once:nnw`) with `#1`, `#2`, ..., `#n` (up to the number of arguments declared for the hook). In that case, if an actual parameter token should be added to the code, it should be doubled.

This is simpler than `\hook_gput_code:nnn`, the code is simply appended to the hook in the order of declaration at the very end, i.e., after all standard code for the hook got executed. Thus if one needs to undo what the standard does one has to do that as part of `<code>`.

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

<code>\hook_gclear_next_code:n</code>	<code>\hook_gclear_next_code:n {\hook}</code>
---------------------------------------	---

Undo any earlier `\hook_gput_next_code:nn`.

<code>\hook_gremove_code:nn</code>	<code>\hook_gremove_code:nn {\hook} {\label}</code>
------------------------------------	---

Removes any code for `<hook>` labeled `<label>`.

If there is no code under the `<label>` in the `<hook>`, or if the `<hook>` does not exist, a warning is issued when you attempt to use `\hook_gremove_code:nn`, and the command is ignored.

If the second argument is `*`, then all code chunks are removed. This is rather dangerous as it drops code from other packages one may not know about, so think twice before using that!

The `<hook>` and `<label>` can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

`\hook_gset_rule:nmmn` `\hook_gset_rule:nmmn` $\langle\mathit{hook}\rangle$ $\langle\mathit{label1}\rangle$ $\langle\mathit{relation}\rangle$ $\langle\mathit{label2}\rangle$

Relate $\langle\mathit{label1}\rangle$ with $\langle\mathit{label2}\rangle$ when used in $\langle\mathit{hook}\rangle$. See `\DeclareHookRule` for the allowed $\langle\mathit{relation}\rangle$ s. If $\langle\mathit{hook}\rangle$ is ?? a default rule is specified.

The $\langle\mathit{hook}\rangle$ and $\langle\mathit{label}\rangle$ can be specified using the dot-syntax to denote the current package name. See section 1.2.1. The dot-syntax is parsed in both $\langle\mathit{label}\rangle$ arguments, but it usually makes sense to be used in only one of them.

`\hook_if_empty_p:n` * `\hook_if_empty:nTF` $\langle\mathit{hook}\rangle$ $\langle\mathit{true\ code}\rangle$ $\langle\mathit{false\ code}\rangle$

`\hook_if_empty:nTF` * Tests if the $\langle\mathit{hook}\rangle$ is empty (*i.e.*, no code was added to it using either `\AddToHook` or `\AddToHookNext`), and branches to either $\langle\mathit{true\ code}\rangle$ or $\langle\mathit{false\ code}\rangle$ depending on the result.

The $\langle\mathit{hook}\rangle$ *cannot* be specified using the dot-syntax. A leading . is treated literally.

`\hook_show:n` `\hook_show:n` $\langle\mathit{hook}\rangle$

`\hook_log:n` `\hook_log:n` $\langle\mathit{hook}\rangle$

Displays information about the $\langle\mathit{hook}\rangle$ such as

- the code chunks (and their labels) added to it,
- any rules set up to order them,
- the computed order in which the chunks are executed,
- any code executed on the next invocation only.

`\hook_log:n` prints the information to the .log file, and `\hook_show:n` prints them to the terminal/command window and starts TeX's prompt (only if `\errorstopmode`) to wait for user action.

The $\langle\mathit{hook}\rangle$ can be specified using the dot-syntax to denote the current package name. See section 1.2.1.

`\hook_debug_on:` `\hook_debug_on:`

`\hook_debug_off:` Turns the debugging of hook code on or off. This displays changes to the hook data.

1.2.3 On the order of hook code execution

Chunks of code for a $\langle\mathit{hook}\rangle$ under different labels are supposed to be independent if there are no special rules set up that define a relation between the chunks. This means that you can't make assumptions about the order of execution!

Suppose you have the following declarations:

```
\NewHook{myhook}
\AddToHook{myhook}[packageA]{\typeout{A}}
\AddToHook{myhook}[packageB]{\typeout{B}}
\AddToHook{myhook}[packageC]{\typeout{C}}
```

then executing the hook with `\UseHook` will produce the typeout A B C in that order. In other words, the execution order is computed to be `packageA`, `packageB`, `packageC` which you can verify with `\ShowHook{myhook}`:

```

-> The hook 'myhook':
> Code chunks:
>   packageA -> \typeout {A}
>   packageB -> \typeout {B}
>   packageC -> \typeout {C}
> Document-level (top-level) code (executed last):
>   ---
> Extra code for next invocation:
>   ---
> Rules:
>   ---
> Execution order:
>   packageA, packageB, packageC.

```

The reason is that the code chunks are internally saved in a property list and the initial order of such a property list is the order in which key-value pairs got added. However, that is only true if nothing other than adding happens!

Suppose, for example, you want to replace the code chunk for `packageA`, e.g.,

```

\RemoveFromHook{myhook}[packageA]
\AddToHook{myhook}[packageA]{\typeout{A alt}}

```

then your order becomes `packageB`, `packageC`, `packageA` because the label got removed from the property list and then re-added (at its end).

While that may not be too surprising, the execution order is also sometimes altered if you add a redundant rule, e.g. if you specify

```

\DeclareHookRule{myhook}{packageA}{before}{packageB}

```

instead of the previous lines we get

```

-> The hook 'myhook':
> Code chunks:
>   packageA -> \typeout {A}
>   packageB -> \typeout {B}
>   packageC -> \typeout {C}
> Document-level (top-level) code (executed last):
>   ---
> Extra code for next invocation:
>   ---
> Rules:
>   packageB|packageA with relation >
> Execution order (after applying rules):
>   packageA, packageC, packageB.

```

As you can see the code chunks are still in the same order, but in the execution order for the labels `packageB` and `packageC` have swapped places. The reason is that, with the rule there are two orders that satisfy it, and the algorithm for sorting happened to pick a different one compared to the case without rules (where it doesn't run at all as there is nothing to resolve). Incidentally, if we had instead specified the redundant rule

```

\DeclareHookRule{myhook}{packageB}{before}{packageC}

```

the execution order would not have changed.

In summary: it is not possible to rely on the order of execution unless there are rules that partially or fully define the order (in which you can rely on them being fulfilled).

1.2.4 The use of “reversed” hooks

You may have wondered why you can declare a “reversed” hook with `\NewReversedHook` and what that does exactly.

In short: the execution order of a reversed hook (without any rules!) is exactly reversed to the order you would have gotten for a hook declared with `\NewHook`.

This is helpful if you have a pair of hooks where you expect to see code added that involves grouping, e.g., starting an environment in the first and closing that environment in the second hook. To give a somewhat contrived example⁴, suppose there is a package adding the following:

```
\AddToHook{env/quote/before}[package-1]{\begin{itshape}}
\AddToHook{env/quote/after} [package-1]{\end{itshape}}
```

As a result, all quotes will be in italics. Now suppose further that another `package-too` makes the quotes also in blue and therefore adds:

```
\usepackage{color}
\AddToHook{env/quote/before}[package-too]{\begin{color}{blue}}
\AddToHook{env/quote/after} [package-too]{\end{color}}
```

Now if the `env/quote/after` hook would be a normal hook we would get the same execution order in both hooks, namely:

```
package-1, package-too
```

(or vice versa) and as a result, would get:

```
\begin{itshape}\begin{color}{blue} ...
\end{itshape}\end{color}
```

and an error message saying that `\begin{color}` was ended by `\end{itshape}`. With `env/quote/after` declared as a reversed hook the execution order is reversed and so all environments are closed in the correct sequence and `\ShowHook` would give us the following output:

```
-> The hook 'env/quote/after':
> Code chunks:
>   package-1 -> \end {itshape}
>   package-too -> \end {color}
> Document-level (top-level) code (executed first):
>   ---
> Extra code for next invocation:
>   ---
> Rules:
>   ---
> Execution order (after reversal):
>   package-too, package-1.
```

If there is a matching default rule (done with `\DeclareDefaultHookRule` or with `??` for the hook name) then this default rule is applied before the reversal so that the order in the reversed hook mirrors the one in the normal hook. However, all rules specific to a hook happen always after the reversal of the execution order, so if you alter the order you will probably have to alter it in both hooks, not just in one, but that depends on the use case.

⁴There are simpler ways to achieve the same effect.

1.2.5 Difference between “normal” and “one-time” hooks

When executing a hook a developer has the choice of using either `\UseHook` or `\UseOneTimeHook` (or their `expl3` equivalents `\hook_use:n` and `\hook_use_once:n`). This choice affects how `\AddToHook` is handled after the hook has been executed for the first time.

With normal hooks adding code via `\AddToHook` means that the code chunk is added to the hook data structure and then used each time `\UseHook` is called.

With one-time hooks it this is handled slightly differently: After `\UseOneTimeHook` has been called, any further attempts to add code to the hook via `\AddToHook` will simply execute the `<code>` immediately.

This has some consequences one needs to be aware of:

- If `<code>` is added to a normal hook after the hook was executed and it is never executed again for one or the other reason, then this new `<code>` will never be executed.
- In contrast if that happens with a one-time hook the `<code>` is executed immediately.

In particular this means that construct such as

```
\AddToHook{myhook}
  { <code-1> \AddToHook{myhook}{<code-2>} <code-3> }
```

works for one-time hooks⁵ (all three code chunks are executed one after another), but it makes little sense with a normal hook, because with a normal hook the first time `\UseHook{myhook}` is executed it would

- execute `<code-1>`,
- then execute `\AddToHook{myhook}{code-2}` which adds the code chunk `<code-2>` to the hook for use on the next invocation,
- and finally execute `<code-3>`.

The second time `\UseHook` is called it would execute the above and in addition `<code-2>` as that was added as a code chunk to the hook in the meantime. So each time the hook is used another copy of `<code-2>` is added and so that code chunk is executed `<# of invocations> - 1` times.

1.2.6 Generic hooks provided by packages

The hook management system also implements a category of hooks that are called “Generic Hooks”. Normally a hook has to be explicitly declared before it can be used in code. This ensures that different packages are not using the same hook name for unrelated purposes—something that would result in absolute chaos. However, there are a number of “standard” hooks where it is unreasonable to declare them beforehand, e.g. each and every command has (in theory) an associated **before** and **after** hook. In such cases, i.e., for command, environment or file hooks, they can be used simply by adding code to them with `\AddToHook`. For more specialized generic hooks, e.g., those provided

⁵This is sometimes used with `\AtBeginDocument` which is why it is supported.

by `babel`, you have to additionally enable them with `\ActivateGenericHook` as explained below.

The generic hooks provided by L^AT_EX are those for `cmd`, `env`, `file`, `include`, `package`, and `class`, and all these are available out of the box: you only have to use `\AddToHook` to add code to them, but you don't have to add `\UseHook` or `\UseOneTimeHook` to your code, because this is already done for you (or, in the case of `cmd` hooks, the command's code is patched at `\begin{document}`, if necessary).

However, if you want to provide further generic hooks in your own code, the situation is slightly different. To do this you should use `\UseHook` or `\UseOneTimeHook`, but *without declaring the hook* with `\NewHook`. As mentioned earlier, a call to `\UseHook` with an undeclared hook name does nothing. So as an additional setup step, you need to explicitly activate your generic hook. Note that a generic hook produced in this way is always a normal hook.

For a truly generic hook, with a variable part in the hook name, such upfront activation would be difficult or impossible, because you typically do not know what kind of variable parts may come up in real documents.

For example, `babel` provides hooks such as `babel/⟨language⟩/afterextras`. However, language support in `babel` is often done through external language packages. Thus doing the activation for all languages inside the core `babel` code is not a viable approach. Instead it needs to be done by each language package (or by the user who wants to use a particular hook).

Because the hooks are not declared with `\NewHook` their names should be carefully chosen to ensure that they are (likely to be) unique. Best practice is to include the package or command name, as was done in the `babel` example above.

Generic hooks defined in this way are always normal hooks (i.e., you can't implement reversed hooks this way). This is a deliberate limitation, because it speeds up the processing considerably.

1.2.7 Hooks with arguments

Sometimes it is necessary to pass contextual information to a hook, and, for one reason or another, it is not feasible to store such information in macros. To serve this purpose, hooks can be declared with arguments, so that the programmer can pass along the data necessary for the code in the hook to function properly.

A hook with arguments works mostly like a regular hook, and most commands that work for regular hooks, also work for hooks that take arguments. The differences are when the hook is declared (`\NewHookWithArguments` is used instead of `\NewHook`), then code can be added with both `\AddToHook` and `\AddToHookWithArguments`, and when the hook is used (`\UseHookWithArguments` instead of `\UseHook`).

A hook with arguments must be declared as such (before it is first used, as all regular hooks) using `\NewHookWithArguments{⟨hook⟩}{⟨number⟩}`. All code added to that hook can then use `#1` to access the first argument, `#2` to access the second, and so forth up to the number of arguments declared. However, it is still possible to add code with references to the arguments of a hook that was not yet declared (we will discuss that later). At their core, hooks are macros, so T_EX's limit of 9 arguments applies, and a low-level T_EX error is raised if you try to reference an argument number that doesn't exist.

To use a hook with arguments, just write `\UseHookWithArguments{<hook>}{<number>}` followed by a braced list of the arguments. For example, if the hook `test` takes three arguments, write:

```
\UseHookWithArguments{test}{3}{arg-1}{arg-2}{arg-3}
```

then, in the `<code>` of the hook, all instances of `#1` will be replaced by `arg-1`, `#2` by `arg-2` and so on. If, at the point of usage, the programmer provides more arguments than the hook is declared to take, the excess arguments are simply ignored by the hook. Behavior is unpredictable⁶ if too few arguments are provided. If the hook isn't declared, `<number>` arguments are removed from the input stream.

Adding code to a hook with arguments can be done with `\AddToHookWithArguments` as well as with the regular `\AddToHook`, to achieve different outcomes. The main difference when it comes to adding code to a hook, in this case, is firstly the possibility of accessing a hook's arguments, of course, and second, how parameter tokens (`#6`) are treated.

Using `\AddToHook` in a hook that takes arguments will work as it does for all other hooks. This allows a package developer to add arguments to a hook that otherwise had none without having to worry about compatibility. This means that, for example:

```
\AddToHook{test}{\def\foo#1{Hello, #1!}}
```

will define the same macro `\foo` regardless if the hook `test` takes arguments or not.

Using `\AddToHookWithArguments` allows the `<code>` added to access the arguments of the hook with `#1`, `#2`, and so forth, up to the number of the arguments declared in the hook. This means that if one wants to add a `#6` to the `<code>` that token must be doubled in the input. The same definition from above, using `\AddToHookWithArguments`, needs to be rewritten:

```
\AddToHookWithArguments{test}{\def\foo##1{Hello, ##1!}}
```

Extending the above example to use the hook arguments, we could rewrite something like (now from declaration to usage, to get the whole picture):

```
\NewHookWithArguments{test}{1}
\AddToHookWithArguments{test}{%
  \typeout{Defining foo with "#1"}
  \def\foo##1{Hello, ##1! Some text after: #1}%
}
\UseHook{test}{Howdy!}
\ShowCommand\foo
```

Running the code above prints in the terminal:

```
Defining foo with "Howdy!"
> \foo=macro:
#1->Hello, #1! Some text after: Howdy!.
```

⁶The hook *will* take the declared number of arguments, and what will happen depends on what was grabbed, and what the hook code does with its arguments.

Note how `##1` in the call to `\AddToHookWithArguments` became `#1`, and the `#1` was replaced by the argument passed to the hook. Should the hook be used again, with a different argument, the definition would naturally change.

It is possible to add code referencing a hook’s arguments before such hook is declared and the number of hooks is fixed. However, if some code is added to the hook, that references more arguments than will be declared for the hook, there will be a low-level `TeX` error about an “Illegal parameter number” at the time the hook is declared, which will be hard to track down because at that point `TeX` can’t know whence the offending code came from. Thus it is important that package writers explicitly document how many arguments (if any) each hook can take, so users of those packages know how many arguments can be referenced, and equally important, what each argument means.

1.2.8 Private `LaTeX` kernel hooks

There are a few places where it is absolutely essential for `LaTeX` to function correctly that code is executed in a precisely defined order. Even that could have been implemented with the hook management (by adding various rules to ensure the appropriate ordering with respect to other code added by packages). However, this makes every document unnecessary slow, because there has to be sorting even though the result is predetermined. Furthermore it forces package writers to unnecessarily add such rules if they add further code to the hook (or break `LaTeX`).

For that reason such code is not using the hook management, but instead private kernel commands directly before or after a public hook with the following naming convention: `\@kernel@before@hook` or `\@kernel@after@hook`. For example, in `\enddocument` you find

```
\UseHook{enddocument}%  
\@kernel@after@enddocument
```

which means first the user/package-accessible `enddocument` hook is executed and then the internal kernel hook. As their name indicates these kernel commands should not be altered by third-party packages, so please refrain from that in the interest of stability and instead use the public hook next to it.⁷

1.2.9 Legacy `LaTeX 2ε` interfaces

`LaTeX 2ε` offered a small number of hooks together with commands to add to them. They are listed here and are retained for backwards compatibility.

With the new hook management, several additional hooks have been added to `LaTeX` and more will follow. See the next section for what is already available.

⁷As with everything in `TeX` there is not enforcement of this rule, and by looking at the code it is easy to find out how the kernel adds to them. The main reason of this section is therefore to say “please don’t do that, this is unconfigurable code!”

`\AtBeginDocument` `\AtBeginDocument` [*label*] {*code*}

If used without the optional argument `<label>`, it works essentially like before, i.e., it is adding `<code>` to the hook `begindocument` (which is executed inside `\begin{document}`). However, all code added this way is labeled with the label `top-level` (see section 1.2.1) if done outside of a package or class or with the package/class name if called inside such a file (see section 1.2.1).

This way one can add code to the hook using `\AddToHook` or `\AtBeginDocument` using a different label and explicitly order the code chunks as necessary, e.g., run some code before or after another package’s code. When using the optional argument the call is equivalent to running `\AddToHook {begindocument} [label] {code}`.

`\AtBeginDocument` is a wrapper around the `begindocument` hook (see section 1.3.2), which is a one-time hook. As such, after the `begindocument` hook is executed at `\begin{document}` any attempt to add `<code>` to this hook with `\AtBeginDocument` or with `\AddToHook` will cause that `<code>` to execute immediately instead. See section 1.2.5 for more on one-time hooks.

For important packages with known order requirement we may over time add rules to the kernel (or to those packages) so that they work regardless of the loading-order in the document.

`\AtEndDocument` `\AtEndDocument` [*label*] {*code*}

Like `\AtBeginDocument` but for the `enddocument` hook.

The few hooks that existed previously in $\text{\LaTeX} 2_{\epsilon}$ used internally commands such as `@begindocumenthook` and packages sometimes augmented them directly rather than working through `\AtBeginDocument`. For that reason there is currently support for this, that is, if the system detects that such an internal legacy hook command contains code it adds it to the new hook system under the label `legacy` so that it doesn’t get lost.

However, over time the remaining cases of direct usage need updating because in one of the future release of \LaTeX we will turn this legacy support off, as it does unnecessary slow down the processing.

1.3 $\text{\LaTeX} 2_{\epsilon}$ commands and environments augmented by hooks

In this section we describe the standard hooks that are now offered by \LaTeX , or give pointers to other documents in which they are described. This section will grow over time (and perhaps eventually move to `usrguide3`).

1.3.1 Generic hooks

As stated earlier, with the exception of generic hooks, all hooks must be declared with `\NewHook` before they can be used. All generic hooks have names of the form “`<type>/<name>/<position>`”, where `<type>` is from the predefined list shown below, and `<name>` is the variable part whose meaning will depend on the `<type>`. The last component, `<position>`, has more complex possibilities: it can always be `before` or `after`; for `env` hooks, it can also be `begin` or `end`; and for `include` hooks it can also be `end`. Each specific hook is documented below, or in `ltxcmdhooks-doc.pdf` or `ltxfilehook-doc.pdf`.

The generic hooks provided by \LaTeX belong to one of the six types:

- env** Hooks executed before and after environments – $\langle name \rangle$ is the name of the environment, and available values for $\langle position \rangle$ are **before**, **begin**, **end**, and **after**;
- cmd** Hooks added to and executed before and after commands – $\langle name \rangle$ is the name of the command, and available values for $\langle position \rangle$ are **before** and **after**;
- file** Hooks executed before and after reading a file – $\langle name \rangle$ is the name of the file (with extension), and available values for $\langle position \rangle$ are **before** and **after**;
- package** Hooks executed before and after loading packages – $\langle name \rangle$ is the name of the package, and available values for $\langle position \rangle$ are **before** and **after**;
- class** Hooks executed before and after loading classes – $\langle name \rangle$ is the name of the class, and available values for $\langle position \rangle$ are **before** and **after**;
- include** Hooks executed before and after `\included` files – $\langle name \rangle$ is the name of the included file (without the `.tex` extension), and available values for $\langle position \rangle$ are **before**, **end**, and **after**.

Each of the hooks above are detailed in the following sections and in linked documentation.

Generic hooks for all environments

Every environment $\langle env \rangle$ has now four associated hooks coming with it:

- env/ $\langle env \rangle$ /before** This hook is executed as part of `\begin` as the very first action, in particular prior to starting the environment group. Its scope is therefore not restricted by the environment.
- env/ $\langle env \rangle$ /begin** This hook is executed as part of `\begin` directly in front of the code specific to the environment start (e.g., the third argument of `\NewDocumentEnvironment` and the second argument of `\newenvironment`). Its scope is the environment body.
- env/ $\langle env \rangle$ /end** This hook is executed as part of `\end` directly in front of the code specific to the end of the environment (e.g., the fourth argument of `\NewDocumentEnvironment` and the third argument of `\newenvironment`).
- env/ $\langle env \rangle$ /after** This hook is executed as part of `\end` after the code specific to the environment end and after the environment group has ended. Its scope is therefore not restricted by the environment.

The hook is implemented as a reversed hook so if two packages add code to **env/ $\langle env \rangle$ /before** and to **env/ $\langle env \rangle$ /after** they can add surrounding environments and the order of closing them happens in the right sequence.

Given that these generic hook names involve `/` as part of their name they would not work if one tries to define an environment using a name that involves a `/`.⁸

Generic environment hooks are never one-time hooks even with environments that are supposed to appear only once in a document.⁹ In contrast to other hooks there is also no need to declare them using `\NewHook`.

⁸Officially, L^AT_EX names for environments should only consist of a sequence of letters, numbers, and the character `*`, i.e., this is not a new restriction.

⁹Thus if one adds code to such hooks after the environment has been processed, it will only be executed if the environment appears again and if that doesn't happen the code will never get executed.

The hooks are only executed if `\begin{env}` and `\end{env}` is used. If the environment code is executed via low-level calls to `\env` and `\endenv` (e.g., to avoid the environment grouping) they are not available. If you want them available in code using this method, you would need to add them yourself, i.e., write something like

```
\UseHook{env/quote/before}\quote
...
\endquote\UseHook{env/quote/after}
```

to add the outer hooks, etc.

Largely for compatibility with existing packages, the following four commands are also available to set the environment hooks; but for new packages we recommend directly using the hook names and `\AddToHook`.

`\BeforeBeginEnvironment` `\BeforeBeginEnvironment` [*label*] {*env*} {*code*}

This declaration adds to the `env/⟨env⟩/before` hook using the *label*. If *label* is not given, the *default label* is used (see section 1.2.1).

`\AtBeginEnvironment` `\AtBeginEnvironment` [*label*] {*env*} {*code*}

This is like `\BeforeBeginEnvironment` but it adds to the `env/⟨env⟩/begin` hook.

`\AtEndEnvironment` `\AtEndEnvironment` [*label*] {*env*} {*code*}

This is like `\BeforeBeginEnvironment` but it adds to the `env/⟨env⟩/end` hook.

`\AfterEndEnvironment` `\AfterEndEnvironment` [*label*] {*env*} {*code*}

This is like `\BeforeBeginEnvironment` but it adds to the `env/⟨env⟩/after` hook.

Generic hooks for commands

Similar to environments there are now (at least in theory) two generic hooks available for any \LaTeX command. These are

`cmd/⟨name⟩/before` This hook is executed at the very start of the command execution.

`cmd/⟨name⟩/after` This hook is executed at the very end of the command body. It is implemented as a reversed hook.

In practice there are restrictions and especially the `after` hook works only with a subset of commands. Details about these restrictions are documented in `ltxcmdhooks-doc.pdf` or with code in `ltxcmdhooks-code.pdf`.

Generic hooks provided by file loading operations

There are several hooks added to \LaTeX 's process of loading file via its high-level interfaces such as `\input`, `\include`, `\usepackage`, `\RequirePackage`, etc. These are documented in `ltxfilehook-doc.pdf` or with code in `ltxfilehook-code.pdf`.

1.3.2 Hooks provided by `\begin{document}`

Until 2020 `\begin{document}` offered exactly one hook that one could add to using `\AtBeginDocument`. Experiences over the years have shown that this single hook in one place was not enough and as part of adding the general hook management system a number of additional hooks have been added at this point. The places for these hooks have been chosen to provide the same support as offered by external packages, such as `etoolbox` and others that augmented `\document` to gain better control.

Supported are now the following hooks (all of them one-time hooks):

`begindocument/before` This hook is executed at the very start of `\document`, one can think of it as a hook for code at the end of the preamble section and this is how it is used by `etoolbox`'s `\AtEndPreamble`.

This is a one-time hook, so after it is executed, all further attempts to add code to it will execute such code immediately (see section 1.2.5).

`begindocument` This hook is added to by using `\AddToHook{begindocument}` or by using `\AtBeginDocument` and it is executed after the `.aux` file has been read and most initialization are done, so they can be altered and inspected by the hook code. It is followed by a small number of further initializations that shouldn't be altered and are therefore coming later.

The hook should not be used to add material for typesetting as we are still in \LaTeX 's initialization phase and not in the document body. If such material needs to be added to the document body use the next hook instead.

This is a one-time hook, so after it is executed, all further attempts to add code to it will execute such code immediately (see section 1.2.5).

`begindocument/end` This hook is executed at the end of the `\document` code in other words at the beginning of the document body. The only command that follows it is `\ignorespaces`.

This is a one-time hook, so after it is executed, all further attempts to add code to it will execute such code immediately (see section 1.2.5).

The generic hooks executed by `\begin` also exist, i.e., `env/document/before` and `env/document/begin`, but with this special environment it is better use the dedicated one-time hooks above.

1.3.3 Hooks provided by `\end{document}`

$\text{\LaTeX} 2_{\epsilon}$ has always provided `\AtEndDocument` to add code to the `\end{document}`, just in front of the code that is normally executed there. While this was a big improvement over the situation in $\text{\LaTeX} 2.09$, it was not flexible enough for a number of use cases and so packages, such as `etoolbox`, `atveryend` and others patched `\enddocument` to add additional points where code could be hooked into.

Patching using packages is always problematical as leads to conflicts (code availability, ordering of patches, incompatible patches, etc.). For this reason a number of additional hooks have been added to the `\enddocument` code to allow packages to add code in various places in a controlled way without the need for overwriting or patching the core code.

Supported are now the following hooks (all of them one-time hooks):

enddocument The hook associated with `\AtEndDocument`. It is immediately called at the beginning of `\enddocument`.

When this hook is executed there may be still unprocessed material (e.g., floats on the deferlist) and the hook may add further material to be typeset. After it, `\clearpage` is called to ensure that all such material gets typeset. If there is nothing waiting the `\clearpage` has no effect.

This is a one-time hook, so after it is executed, all further attempts to add code to it will execute such code immediately (see section 1.2.5).

enddocument/afterlastpage As the name indicates this hook should not receive code that generates material for further pages. It is the right place to do some final housekeeping and possibly write out some information to the `.aux` file (which is still open at this point to receive data, but since there will be no more pages you need to write to it using `\immediate\write`). It is also the correct place to set up any testing code to be run when the `.aux` file is re-read in the next step.

After this hook has been executed the `.aux` file is closed for writing and then read back in to do some tests (e.g., looking for missing references or duplicated labels, etc.).

This is a one-time hook, so after it is executed, all further attempts to add code to it will execute such code immediately (see section 1.2.5).

enddocument/afteraux At this point, the `.aux` file has been reprocessed and so this is a possible place for final checks and display of information to the user. However, for the latter you might prefer the next hook, so that your information is displayed after the (possibly longish) list of files if that got requested via `\listfiles`.

This is a one-time hook, so after it is executed, all further attempts to add code to it will execute such code immediately (see section 1.2.5).

enddocument/info This hook is meant to receive code that write final information messages to the terminal. It follows immediately after the previous hook (so both could have been combined, but then packages adding further code would always need to also supply an explicit rule to specify where it should go.

This hook already contains some code added by the kernel (under the labels `kernel/filelist` and `kernel/warnings`), namely the list of files when `\listfiles` has been used and the warnings for duplicate labels, missing references, font substitutions etc.

This is a one-time hook, so after it is executed, all further attempts to add code to it will execute such code immediately (see section 1.2.5).

enddocument/end Finally, this hook is executed just in front of the final call to `\@@end`.

This is a one-time hook, so after it is executed, all further attempts to add code to it will execute such code immediately (see section 1.2.5).is it even possible to add code after this one?

There is also the hook `shipout/lastpage`. This hook is executed as part of the last `\shipout` in the document to allow package to add final `\special`'s to that page. Where this hook is executed in relation to those from the above list can vary from document to document. Furthermore to determine correctly which of the `\shipouts` is the last one,

L^AT_EX needs to be run several times, so initially it might get executed on the wrong page. See section 1.3.4 for where to find the details.

It is also possible to use the generic `env/document/end` hook which is executed by `\end`, i.e., just in front of the first hook above. Note however that the other generic `\end` environment hook, i.e., `env/document/after` will never get executed, because by that time L^AT_EX has finished the document processing.

1.3.4 Hooks provided by `\shipout` operations

There are several hooks and mechanisms added to L^AT_EX's process of generating pages. These are documented in `ltshipout-doc.pdf` or with code in `ltshipout-code.pdf`.

1.3.5 Hooks provided for paragraphs

The paragraph processing has been augmented to include a number of internal and public hooks. These are documented in `ltpara-doc.pdf` or with code in `ltpara-code.pdf`.

1.3.6 Hooks provided in NFSS commands

In languages that need to support for more than one script in parallel (and thus several sets of fonts, e.g., supporting both Latin and Japanese fonts), NFSS font commands such as `\sffamily` need to switch both the Latin family to “Sans Serif” and in addition alter a second set of fonts.

To support this, several NFSS commands have hooks to which such support can be added.

rmfamily After `\rmfamily` has done its initial checks and prepared a font series update, this hook is executed before `\selectfont`.

sffamily This is like the `rmfamily` hook, but for the `\sffamily` command.

ttfamily This is like the `rmfamily` hook, but for the `\ttfamily` command.

normalfont The `\normalfont` command resets the font encoding, family, series and shape to their document defaults. It then executes this hook and finally calls `\selectfont`.

expand@font@defaults The internal `\expand@font@defaults` command expands and saves the current defaults for the metafamilies (`rm/sf/tt`) and the metaseries (`bf/md`). If the NFSS machinery has been augmented, e.g., for Chinese or Japanese fonts, then further defaults may need to be set at this point. This can be done in this hook which is executed at the end of this macro.

bfseries/defaults, bfseries If the `\bfdefault` was explicitly changed by the user, its new value is used to set the `bf` series defaults for the metafamilies (`rm/sf/tt`) when `\bfseries` is called. The `bfseries/defaults` hook allows further adjustments to be made in this case. This hook is only executed if such a change is detected. In contrast, the `bfseries` hook is always executed just before `\selectfont` is called to change to the new series.

mdseries/defaults, mdseries These two hooks are like the previous ones but they are in the `\mdseries` command.

selectfont This hook is executed inside `\selectfont`, after the current values for *encoding*, *family*, *series*, *shape*, and *size* are evaluated and the new font is selected (and if necessary loaded). After the hook has executed, NFSS will still do any updates necessary for a new *size* (such as changing the size of `\strut`) and any updates necessary to a change in *encoding*.

This hook is intended for use cases where, in parallel to a change in the main font, some other fonts need to be altered (e.g., in CJK processing where you may need to deal with several different alphabets).

1.3.7 Hook provided by the mark mechanism

See `ltmarks-doc.pdf` for details.

insertmark This hook allows for a special setup while `\InsertMark` inserts a mark. It is executed in group so local changes only apply to the mark being inserted.

Chapter 2

L^AT_EX's hook management for files

2.1 Introduction

2.1.1 Provided hooks

The code offers a number of hooks into which packages (or the user) can add code to support different use cases. Many hooks are offered as pairs (i.e., the second hook is reversed). Also important to know is that these pairs are properly nested with respect to other pairs of hooks.

There are hooks that are executed for all files of a certain type (if they contain code), e.g., for all “include files” or all “packages”, and there are also hooks that are specific to a single file, e.g., do something after the package `foo.sty` has been loaded.

2.1.2 General hooks for file reading

There are four hooks that are called for each file that is read using document-level commands such as `\input`, `\include`, `\usepackage`, etc. They are not called for files read using internal low-level methods, such as `\@input` or `\openin`.

<code>file/before</code>
<code>file/.../before</code>
<code>file/.../after</code>
<code>file/after</code>

These are:

file/before, file/⟨file-name⟩/before These hooks are executed in that order just before the file is loaded for reading. The code of the first hook is used with every file, while the second is executed only for the file with matching `⟨file-name⟩` allowing you to specify code that only applies to one file.

file/⟨file-name⟩/after, file/after These hooks are executed after the file with name `⟨file-name⟩` has been fully consumed. The order is swapped (the specific one comes first) so that the `/before` and `/after` hooks nest properly, which is important if any of them involve grouping (e.g., contain environments, for example). Furthermore both hooks are reversed hooks to support correct nesting of different packages adding code to both `/before` and `/after` hooks.

So the overall sequence of hook processing for any file read through the user interface commands of L^AT_EX is:

```

\UseHook{file/before}
\UseHook{file/<file name>/before}
  <file contents>
\UseHook{file/<file name>/after}
\UseHook{file/after}

```

The file hooks only refer to the file by its name and extension, so the *<file name>* should be the file name as it is on the filesystem with extension (if any) and without paths. Different from `\input` and similar commands, the `.tex` extension is not assumed in hook *<file name>*, so `.tex` files must be specified with their extension to be recognized. Files within subfolders should also be addressed by their name and extension only.

Extensionless files also work, and should then be given without extension. Note however that T_EX prioritizes `.tex` files, so if two files `foo` and `foo.tex` exist in the search path, only the latter will be seen.

When a file is input, the *<file name>* is available in `\CurrentFile`, which is then used when accessing the `file/<file name>/before` and `file/<file name>/after`.

`\CurrentFile` The name of the file about to be read (or just finished) is available to the hooks through `\CurrentFile` (there is no `expl3` name for it for now). The file is always provided with its extension, i.e., how it appears on your hard drive, but without any specified path to it. For example, `\input{sample}` and `\input{app/sample.tex}` would both have `\CurrentFile` being `sample.tex`.

`\CurrentFilePath` The path to the current file (complement to `\CurrentFile`) is available in `\CurrentFilePath` if needed. The paths returned in `\CurrentFilePath` are only user paths, given through `\input@path` (or `expl3`'s equivalent `\l_file_search_path_seq`) or by directly typing in the path in the `\input` command or equivalent. Files located by `kpsewhich` get the path added internally by the T_EX implementation, so at the macro level it looks as if the file were in the current folder, so the path in `\CurrentFilePath` is empty in these cases (package and class files, mostly).

`\CurrentFileUsed`
`\CurrentFilePathUsed` In normal circumstances these are identical to `\CurrentFile` and `\CurrentFilePath`. They will differ when a file substitution has occurred for `\CurrentFile`. In that case, `\CurrentFileUsed` and `\CurrentFilePathUsed` will hold the actual file name and path loaded by L^AT_EX, while `\CurrentFile` and `\CurrentFilePath` will hold the names that were *asked for*. Unless doing very specific work on the file being read, `\CurrentFile` and `\CurrentFilePath` should be enough.

2.1.3 Hooks for package and class files

Commands to load package and class files (e.g., `\usepackage`, `\RequirePackage`, `\LoadPackageWithOptions`, etc.) offer the hooks from section 2.1.2 when they are used to load a package or class file, e.g., `file/array.sty/after` would be called after the `array` package got loaded. But as packages and classes form as special group of files, there are some additional hooks available that only apply when a package or class is loaded.

package/before	These are:
package/after	
package/.../before	package/before, package/after These hooks are called for each package being loaded.
package/.../after	
class/before	package/⟨name⟩/before, package/⟨name⟩/after These hooks are additionally called if the package name is ⟨name⟩ (without extension).
class/after	
class/.../before	class/before, class/after These hooks are called for each class being loaded.
class/.../after	
	class/⟨name⟩/before, class/⟨name⟩/after These hooks are additionally called if the class name is ⟨name⟩ (without extension).

All /after hooks are implemented as reversed hooks.

The overall sequence of execution for `\usepackage` and friends is:

```

\UseHook{package/before}
\UseOneTimeHook{package/⟨package name⟩/before}
  \UseHook{file/before}
  \UseHook{file/⟨package name⟩.sty/before}
    ⟨package contents⟩
  \UseHook{file/⟨package name⟩.sty/after}
  \UseHook{file/after}
  code from \AtEndOfPackage if used inside the package
\UseOneTimeHook{package/⟨package name⟩/after}
\UseHook{package/after}

```

and similar for class file loading, except that `package/` is replaced by `class/` and `\AtEndOfPackage` by `\AtEndOfClass`.

If a package or class is not loaded none of the hooks are executed!

All class or package hooks involving the name of the class or package are implemented as one-time hooks, whereas all other such hooks are normal hooks. This allows for the following use case

```

\AddToHook{package/varioref/after}
{ ... apply my customizations if the package gets
  loaded (or was loaded already) ... }

```

without the need to first test if the package is already loaded.

2.1.4 Hooks for `\include` files

To manage `\include` files, L^AT_EX issues a `\clearpage` before and after loading such a file. Depending on the use case one may want to execute code before or after these `\clearpages` especially for the one that is issued at the end.

Executing code before the final `\clearpage`, means that the code is processed while the last page of the included material is still under construction. Executing code after it means that all floats from inside the include file are placed (which might have added further pages) and the final page has finished.

Because of these different scenarios we offer hooks in three places.¹⁰ None of the hooks are executed when an `\include` file is bypassed because of an `\includeonly` declaration. They are, however, all executed if L^AT_EX makes an attempt to load the `\include` file (even if it doesn't exist and all that happens is “No file `\langle filename \rangle.tex`”).

```
include/before
include/.../before
include/end
include/.../end
include/after
include/.../after
```

These are:

include/before, include/`\langle name \rangle`/before These hooks are executed (in that order) after the initial `\clearpage` and after `.aux` file is changed to use `\langle name \rangle.aux`, but before the `\langle name \rangle.tex` file is loaded. In other words they are executed at the very beginning of the first page of the `\include` file.

include/`\langle name \rangle`/end, include/end These hooks are executed (in that order) after L^AT_EX has stopped reading from the `\include` file, but before it has issued a `\clearpage` to output any deferred floats.

include/`\langle name \rangle`/after, include/after These hooks are executed (in that order) after L^AT_EX has issued the `\clearpage` but before it has switched back writing to the main `.aux` file. Thus technically we are still inside the `\include` and if the hooks generate any further typeset material including anything that writes to the `.aux` file, then it would be considered part of the included material and bypassed if it is not loaded because of some `\includeonly` statement.¹¹

include/excluded, include/`\langle name \rangle`/excluded The above hooks for `\include` files are only executed when the file is loaded (or more exactly the load is attempted). If, however, the `\include` file is explicitly excluded (through an `\includeonly` statement) the above hooks are bypassed and instead the `include/excluded` hook followed by the `include/\langle name \rangle/excluded` hook are executed. This happens after L^AT_EX has loaded the `.aux` file for this include file, i.e., after L^AT_EX has updated its counters to pretend that the file was seen.

All `include` hooks involving the name of the included file are implemented as one-time hooks (whereas all other such hooks are normal hooks).

If you want to execute code that is run for every `\include` regardless of whether or not it is excluded, use the `cmd/include/before` or `cmd/include/after` hooks.

2.1.5 High-level interfaces for L^AT_EX

We do not provide any additional wrappers around the hooks (like `filehook` or `scrfile` do) because we believe that for package writers the high-level commands from the hook management, e.g., `\AddToHook`, etc. are sufficient and in fact easier to work with, given that the hooks have consistent naming conventions.

¹⁰If you want to execute code before the first `\clearpage` there is no need to use a hook—you can write it directly in front of the `\include`.

¹¹For that reason another `\clearpage` is executed after these hooks which normally does nothing, but starts a new page if further material got added this way.

2.1.6 Kernel, class, and package interfaces for L^AT_EX

<code>\declare@file@substitution</code>	<code>\declare@file@substitution {<file>} {<replacement-file>}</code>
<code>\undeclare@file@substitution</code>	<code>\undeclare@file@substitution {<file>}</code>

If `<file>` is requested for loading replace it with `<replacement-file>`. `\CurrentFile` remains pointing to `<file>` but `\CurrentFileUsed` will show the file actually loaded.

The main use case for this declaration is to provide a corrected version of a package that can't be changed (due to its license) but no longer functions because of L^AT_EX kernel changes, for example, or to provide a version that makes use of new kernel functionality while the original package remains available for use with older releases. As such it is mainly meant for use in the L^AT_EX kernel but other use cases are conceivable.

The `\undeclare@file@substitution` declaration undoes a substitution made earlier.

Please do not misuse this functionality and replace a file with another unless if really needed and only if the new version is implementing the same functionality as the original one!

<code>\disable@package@load</code>	<code>\disable@package@load {<package>} {<alternate-code>}</code>
<code>\reenable@package@load</code>	<code>\reenable@package@load {<package>}</code>

If `<package>` is requested, do not load it but instead run `<alternate-code>` which could issue a warning, error or any other code.

The main use case is for classes that want to restrict the set of supported packages or contain code that make the use of some packages impossible. So rather than waiting until the document breaks they can set up informative messages why certain packages are not available.

The function is only implemented for packages not for arbitrary files and again it should only be applied if there are good reasons for doing this.¹²

2.1.7 A sample package for structuring the log output

As an application we provide the package `structuredlog` that adds lines to the `.log` when a file is opened and closed for reading keeping track of nesting level as well. For example, for the current document it adds the lines

```
= (LEVEL 1 START) t1lmr.fd
= (LEVEL 1 STOP) t1lmr.fd
= (LEVEL 1 START) supp-pdf.mkii
= (LEVEL 1 STOP) supp-pdf.mkii
= (LEVEL 1 START) nameref.sty
== (LEVEL 2 START) refcount.sty
== (LEVEL 2 STOP) refcount.sty
== (LEVEL 2 START) gettitlestring.sty
== (LEVEL 2 STOP) gettitlestring.sty
= (LEVEL 1 STOP) nameref.sty
= (LEVEL 1 START) ltfilehook-doc.out
```

¹²Just to be sure: “I don't like this package by somebody else” is not a good one :-)

```

= (LEVEL 1 STOP) ltfilehook-doc.out
= (LEVEL 1 START) ltfilehook-doc.out
= (LEVEL 1 STOP) ltfilehook-doc.out
= (LEVEL 1 START) ltfilehook-doc.hd
= (LEVEL 1 STOP) ltfilehook-doc.hd
= (LEVEL 1 START) ltfilehook.dtx
== (LEVEL 2 START) ot1lmr.fd
== (LEVEL 2 STOP) ot1lmr.fd
== (LEVEL 2 START) omllmm.fd
== (LEVEL 2 STOP) omllmm.fd
== (LEVEL 2 START) omslmsy.fd
== (LEVEL 2 STOP) omslmsy.fd
== (LEVEL 2 START) omxmlmex.fd
== (LEVEL 2 STOP) omxmlmex.fd
== (LEVEL 2 START) umsa.fd
== (LEVEL 2 STOP) umsa.fd
== (LEVEL 2 START) umsb.fd
== (LEVEL 2 STOP) umsb.fd
== (LEVEL 2 START) ts1lmr.fd
== (LEVEL 2 STOP) ts1lmr.fd
== (LEVEL 2 START) t1lmss.fd
== (LEVEL 2 STOP) t1lmss.fd
= (LEVEL 1 STOP) ltfilehook.dtx

```

Thus if you inspect an issue in the .log it is easy to figure out in which file it occurred, simply by searching back for LEVEL and if it is a STOP then remove 1 from the level value and search further for LEVEL with that value which should then be the START level of the file you are in.

Chapter 3

Hook management for commands

3.1 Introduction

This file implements generic hooks for (arbitrary) commands. In theory every command `\<name>` offers now two associated hooks to which code can be added using `\AddToHook`,¹³ `\AddToHookNext`, `\AddToHookWithArguments`, and `\AddToHookNextWithArguments`.¹⁴

However, this is only true “in theory”. In practice there are a number of restrictions that makes it impossible to use such generic command hooks in a number of cases, so please read all of section 3.2 to understand what may prevent you from using them successfully.

The generic command hooks are:

cmd/<name>/before This hook is executed at the very start of the command, right after its arguments (if any) are parsed. The hook `<code>` runs in the command inside a call to `\UseHookWithArguments`. Any code added to this hook using `\AddToHookWithArguments` or `\AddToHookNextWithArguments` can access the command’s arguments using `#1`, `#2`, etc., up to the number of arguments of the command. If `\AddToHook` or `\AddToHookNext` are used, the arguments cannot be accessed (see the `lthooks` documentation¹⁵ on hooks with arguments).

cmd/<name>/after This hook is similar to `cmd/<name>/before`, but it is executed at the very end of the command body. This hook is implemented as a reversed hook.

The hooks are not physically present before `\begin{document}`¹⁶ (i.e., using a command in the preamble will never execute the hook) and if nobody has declared any code for them, then they are not added to the command code ever. For example, if we have the following definition

¹³In this documentation, when something is being said about `\AddToHook`, the same will be valid for `\AddToHookWithArguments`, unless that particular paragraph is highlighting the differences between both. The same is true for the other hook-related functions and their `...WithArguments` counterparts.

¹⁴In practice this is not supported for all types of commands, see section 3.2.4 for the restrictions that apply and what happens if one tries to use this with commands for which this is not supported.

¹⁵`texdoc lthooks-doc`

¹⁶More specifically, they are inserted in the commands after the `begindocument` hook, so they are also not present while `LATEX` is reading the `.aux` file.

```
\newcommand\foo[2]{Code #1 for #2!}
```

then executing `\foo{A}{B}` will simply run `Code_A_for_B!` as it was always the case. However, if somebody, somewhere (e.g., in a package) adds

```
\AddToHook{cmd/foo/before}{<before code>}
```

then, after `\begin{document}` the definition of `\foo` will be:

```
\renewcommand\foo[2]{%
  \UseHookWithArguments{cmd/foo/before}{2}{#1}{#2}%
  Code #1 for #2!}
```

and similarly `\AddToHook{cmd/foo/after}{<after code>}` alters the definition to

```
\renewcommand\foo[2]{%
  Code #1 for #2!%
  \UseHookWithArguments{cmd/foo/after}{2}{#1}{#2}}
```

In other words, the mechanism is similar to what `etoolbox` offers with `\pretocmd` and `\apptocmd` with the important differences

- that code can be prepended or appended (i.e., added to the hooks) even if the command itself is not (yet) defined, because the defining package has not been loaded at this point;
- and that by using the hook management interface it is now possible to define how the code chunks added in these places are ordered, if different packages want to add code at these points.

3.2 Restrictions and operational details

Adding arbitrary material to commands is tricky because most of the time we do not know what the macro expects as arguments when expanding and `TEX` doesn't have a reliable way to see that, so some guesswork has to be employed.

We can do this in most cases when commands are defined using `\NewDocumentCommand` or `\newcommand` (with a few exceptions). For commands defined with `\def` the situation is less good. Common cases where the command hooks will not work are:

- Commands that use special catcode settings within their definition. In that case it is usually not possible to augment the definition (see 3.2.1).
- If a command is defined while `\ExplSyntaxOn` is in force **and** the command contains `~` characters to represent spaces, then it can't be patched to include the command hooks. In fact in some very special circumstances you might even get a low-level error rather than the information that the command can't be patched (see, for example, <https://github.com/latex3/latex2e/issues/1430>).
- Commands that have arguments as far as the user is concerned (e.g., `\section` or `\caption`), but are defined in a way that these arguments are not read by the user level command but only later during the processing. In that case the **after** hook doesn't work at all. The before hook only works with `\AddToHook` but not with `\AddToHookWithArguments` because the arguments haven't been read at that point where the hook is patched in. See section 3.2.4.

- Adding a specific generic command hook is only attempted once per command, thus after redefining a command such hooks will no longer be there and will also not being re-added, see section 3.2.1.

All this means that you have to have a good understanding of how commands are defined when you attempt to make use of such hooks and something goes wrong. What can help in that case is to turn on `\DebugHooksOn` in which case you get much more (low-level) details on why something fails and what was tried to enable the hooks.

3.2.1 Patching

The code here tries to find out if a command was defined with `\newcommand` or `\DeclareRobustCommand` or `\NewDocumentCommand`, and if so it *assumes* that the argument specification of the command is as expected (which is not fail-proof, if someone redefines the internals of these commands in devious ways, but is a reasonable assumption).

If the command is one of the defined types, the code here does a sandboxed expansion of the command such that it can be redefined again exactly as before, but with the hook code added.

If however the command is not a known type (it was defined with `\def`, for example), then the code uses an approach similar to `etoolbox`'s `\patchcmd` to retokenize the command with the hook code in place. This procedure, however, is more likely to fail if the `catcode` settings are not the same as the ones at the time of command's definition, so not always adding a hook to a command will work.

Timing

When `\AddToHook` (or its `expl3` equivalent) is called with a generic `cmd` hook, say, `cmd/foo/before`, for the first time (that is, no code was added to that same hook before), in the preamble of a document, it will store a patch instruction for that command until `\begin{document}`, and only then all the commands which had hooks added will be patched in one go. That means that no command in the preamble will have hooks patched into them.

At `\begin{document}` all the delayed patches will be executed, and if the command doesn't exist the code is still added to the hook, but it will not be executed. After `\begin{document}`, when `\AddToHook` is called with a generic `cmd` hook the first time, the command will be immediately patched to include the hook, and if it doesn't exist or if it can't be patched for any reason, an error is thrown; if `\AddToHook` was already used in the preamble no new patching is attempted.

This has the consequence that a command defined or redefined after `\begin{document}` only uses generic `cmd` hook code if `\AddToHook` is called for the first time after the definition is made, or if the command explicitly uses the generic hook in its definition by declaring it with `\NewHookPair` adding `\UseHook` as part of the code.¹⁷

3.2.2 Command copies

Once a hook has been added to a command, it will be present in any copies of that command which are made. For example, if in the preamble we have

¹⁷We might change this behavior in the main document slightly after gaining some usage experience.

```
\NewDocumentCommand\foo{}{}
\AddToHook{cmd/foo/after}{1}%
```

then after `\begin{document}`, any use of `\NewCommandCopy` will include the hook in the copy, for example

```
\NewCommandCopy\baz\foo
```

would include the hook in this copied command.

3.2.3 Grouping

Command hooks are intended to be added to document commands, which are typically defined globally. As such, adding a hook is a global operation, even if the command was previously only locally-defined. Addition of material to hooks is also global.

3.2.4 Commands that look ahead

Some commands are defined in different “steps” and they look ahead in the input stream to find more arguments. If you try to add some code to the `cmd/<name>/after` hook of such command, it will not work, and it is not possible to detect that programmatically, so the user has to know (or find out) which commands can or cannot have hooks attached to them.

One good example is the `\section` command. You can add something to the `cmd/section/before` hook (but only with `\AddToHook` not `\AddToHookWithArguments`), but if you try to add anything to the `cmd/section/after` hook, `\section` will no longer work at all. That happens because the `\section` macro takes no argument, but instead calls a few internal \LaTeX macros to look for the optional and mandatory arguments. By adding code to the `cmd/section/after` hook, you get in the way of that scanning.

In such a case, where it is known that a specific generic command hook does not work if code is added to it, the package author can add a `\DisableGenericHook`¹⁸ declaration to prevent this from happening in user documents and thereby avoiding obscure errors.

3.3 Package author interface

The `cmd` hooks are, by default, available for all commands that can be patched to add the hooks. For some commands, however, the very beginning or the very end of the code is not the best place to put the hooks, for example, if the command looks ahead for arguments (see section 3.2.4).

If you are a package author and you want to add the hooks to your own commands in the proper position you can define the command and manually add the `\UseHookWithArguments` calls inside the command in the proper positions, and manually define the hooks with `\NewHookWithArguments` or `\NewReversedHookWithArguments`. When the hooks are explicitly defined, patching is not attempted so you can make sure your command works properly. For example, an (admittedly not really useful) command that typesets its contents in a framed box with width optionally given in parentheses:

```
\newcommand\fancybox{\@ifnextchar({\@fancybox}{\@fancybox(5cm)}}
\def\@fancybox(#1)#2{\fbox{\parbox{#1}{#2}}}
```

¹⁸Please use `\DisableGenericHook` if at all, only on hooks that you “own”, i.e., for commands your package or class defines and not second guess whether or not hooks of other packages should get disabled!

If you try that definition, then add some code after it with

```
\AddToHook{cmd/fancybox/after}{<code>}
```

and then use the `\fancybox` command you will see that it will be completely broken, because the hook will get executed in the middle of parsing for optional `(...)` argument.

If, on the other hand, you want to add hooks to your command you can do something like:

```
\newcommand\fancybox{\@ifnextchar({\@fancybox}{\@fancybox(5cm)}}
\def\@fancybox(#1)#2{\fbox{%
    \UseHookWithArguments{cmd/fancybox/before}{2}{#1}{#2}%
    \parbox{#1}{#2}%
    \UseHookWithArguments{cmd/fancybox/after}{2}{#1}{#2}}}
\NewHookWithArguments{cmd/fancybox/before}{2}
\NewReversedHookWithArguments{cmd/fancybox/after}{2}
```

then the hooks will be executed where they should and no patching will be attempted. It is important that the hooks are declared with `\NewHookWithArguments` or `\NewReversedHookWithArguments`, otherwise the command hook code will try to patch the command. Note also that the call to `\UseHookWithArguments{cmd/fancybox/before}` does not need to be in the definition of `\fancybox`, but anywhere it makes sense to insert it (in this case in the internal `\@fancybox`).

Alternatively, if for whatever reason your command does not support the generic hooks provided here, you can disable a hook with `\DisableGenericHook`¹⁹, so that when someone tries to add code to it they will get an error. Or if you don't want the error, you can simply declare the hook with `\NewHook` and never use it.

The above approach is useful for really complex commands where for one or the other reason the hooks can't be placed at the very beginning and end of the command body and some hand-crafting is needed. However, in the example above the real (and in fact only) issue is the cascading argument parsing in the style developed long ago in L^AT_EX 2.09. Thus, a much simpler solution for this case is to replace it with the modern `\NewDocumentCommand` syntax and define the command as follows:

```
\DeclareDocumentCommand\fancybox{D()}{5cm}m}{\fbox{\parbox{#1}{#2}}}
```

If you do that then both hooks automatically work and are patched into the right places.

3.3.1 Arguments and redefining commands

The code in `ltxcmds` does its best to find out how many arguments a given command has, and to insert the appropriate call to `\UseHookWithArguments`, so that the arguments seen by the hook are exactly those grabbed by the command (the hook, after all, is a macro call, so the arguments have to be placed in the right order, or they won't match).

When using the package writer interface, as discussed in section 3.3, to change the position of the hooks in your commands, you are also free to change how the hook code in your command sees its arguments. When a `cmd` hook is declared with `\NewHook` (or `\NewHookWithArguments` or other variations of that), it loses its “generic” nature and works as a regular hook. This means that you may choose to declare it without arguments

¹⁹Please use `\DisableGenericHook` if at all, only on hooks that you “own”, i.e., for commands your package or class defines and not second guess whether or not hooks of other packages should get disabled!

regardless if the command takes arguments or not, or declare it with arguments, even if the command takes none.

However, this flexibility should not be abused. When using a nonstandard configuration for the hook arguments, think reasonably: a user will expect that the argument #1 in the hook corresponds to the argument's first argument, and so on. Any other configuration is likely to cause confusion and, if used, will have to be well documented.

This flexibility, however, allows you to “correct” the arguments for the hooks. For example, L^AT_EX's `\refstepcounter` has a single argument, the name of the counter. The `cleveref` package adds an optional argument to `\refstepcounter`, making the name of the counter argument #2. If the author of `cleveref` wanted, for whatever reason, to add hooks to `\refstepcounter`, to preserve compatibility he could write something along the lines of:

```
\NewHookWithArguments{cmd/refstepcounter/before}{1}
\renewcommand\refstepcounter[2][<default>]{%
  \UseHookWithArguments{cmd/refstepcounter/before}{1}{#2}%
  <code for \refstepcounter>}
```

so that the mandatory argument, which is arg #2 in the definition, would still be seen as #1 in the hook code.

Another possibility would be to place the optional argument as the second argument for the hook, so that people looking for it would be able to use it. In either case, it would have to be well documented to cause as little confusion as possible.

Chapter 4

Paragraph building and hooks

4.1 Introduction

The building of paragraphs in the $\text{T}_{\text{E}}\text{X}$ engine(s) has a number of peculiarities that makes it on one hand fairly flexible but on the other hand somewhat awkward to control or reliably to extend. Thus to better understand the code below we start with a brief introduction of the mechanism; for more details refer to the $\text{T}_{\text{E}}\text{X}$ book [?, chap. 14] (for the full truth you may even have to study the program code).

4.1.1 The default processing done by the engine

$\text{T}_{\text{E}}\text{X}$ automatically starts building a paragraph when it is currently in vertical mode and encounters anything that can only live in horizontal mode. Most often this is a character, but there are also many commands that can be used only in horizontal mode. If any of them is encountered, $\text{T}_{\text{E}}\text{X}$ will immediately back up (i.e., the character or command is read later again), adds a `\parskip` glue to the current vertical list unless the list is empty, switches to horizontal mode, starts its special “start of paragraph processing” and only then rereads the character or command that caused the mode change.²⁰

This “start of paragraph processing” first adds an empty box at the start of the horizontal list of width `\parindent` (which represents the paragraph indentation) unless the paragraph was started with `\noindent` in which case no such box is added²¹. It then reads and processes all tokens stored in the special engine token register `\everypar`. After that it reads and processes whatever has caused the paragraph to start.

Thus out of the box, $\text{T}_{\text{E}}\text{X}$ offers the possibility to put some special code into `\everypar` to gain control at (more or less) the start of the paragraph. For example, in LaTeX and a number of packages, special code like the following is sometimes used:

```
\everypar{\setbox\z@\lastbox}\everypar{ ...}
```

This removes the paragraph indentation box again (that was already placed by $\text{T}_{\text{E}}\text{X}$), then resets `\everypar` so that it doesn't do anything on the next paragraph start and then does whatever it wants to do, e.g., in an `\item` of a list it will typeset the label in front of the paragraph text. However, there is only one such `\everypar` token register and if

²⁰Already not quite true: the command `\noindent` starts the paragraph but influences the special processing by suppressing the paragraph indentation box normally inserted by it.

²¹That's a bit different from placing a zero-sized box!

different packages and/or the kernel all attempt to add their own code here, coordination is very difficult if not impossible.

The process when the paragraph ends has different mechanisms and interfaces. A paragraph ends when the engine primitive `\par` is called while `TeX` is in unrestricted horizontal mode, i.e., is building a paragraph. At other times this primitive does nothing or generates an error depending on the mode `TeX` is in, e.g., the `\par` in `\hbox{a\par b}` is ignored, but `$a\par b$` would complain.

If this primitive ends the paragraph it does some special “end of horizontal list” processing, then calls `TeX`’s paragraph builder; this breaks the horizontal list into lines and then these lines are added as boxes to the enclosing vertical list and `TeX` returns to vertical mode.

This `\par` command can be given explicitly, but there are also situations in which `TeX` is generating it on the fly. Most often this happens when `TeX` encounters a blank line which is automatically changed to a `\par` command which is then executed. The other possibility is that `TeX` encounters a command which is incompatible with horizontal processing, e.g., `\vskip` (a request for adding vertical space). In such cases it silently backs up, and inserts a `\par` in the hope that this gets it out of horizontal mode and makes the vertical command acceptable.

The important point to note here is that `TeX` really inserts the command with the name `\par`, which can be redefined. Thus, it may not have its original “primitive” meaning and therefore may not end the horizontal list and call the paragraph builder. This approach offers some flexibility but also allows you to easily produce a `TeX` document that loops forever, for example, the simple line

```
A \let\par\relax \vskip
```

will start a horizontal list at `A`, redefines `\par`, then sees `\vskip` and inserts `\par` to end the paragraph. But this now only runs `\relax` so nothing changes and `\vskip` is read again, issues a `\par` which In short, it only takes a plain `TeX` document with five tokens to run forever (since no memory is consumed and therefore eventually exhausted).

There is no way other than changing `\par` to gain control at the end of a paragraph, i.e., there is no token list like `\everypar` that is inserted. Hence the only way to change the default behavior is to modify the action that `\par` executes, with similar issues as outlined before: different processes need to ensure that they do not overwrite their modifications or worse, think that the `\par` in front of them is the engine primitive while in fact it has already been changed by other code.

To make matters slightly worse there are a few places where `TeX` handles the situation differently (most likely for speed reasons back when computers were much slower). If `TeX` finds itself in unrestricted horizontal mode at the end of building a vertical box (for an `\insert`, `\vadjust` or executing the output routine code), it will finish the horizontal list not by issuing a `\par` command (which would be consistent with all other places) but by simply executing the primitive meaning of `\par`, regardless of the actual definition that `\par` has at the time.

Thus, if you have carefully crafted a redefined `\par` to execute some special actions at the end of a paragraph and you write something like

```
\vbox{Some paragraph ... text.}
```

you will find that your code does not get run for the last paragraph in that box. `LATeX` avoids this problem, by making sure that its boxes (such as `\parbox` or the `minipage` environment, etc.) all internally add an explicit `\par` at the end so that such code is run

and \TeX finds itself in vertical mode already without the need to start up the paragraph builder internally. But, of course, this only works for boxes under direct control of the \LaTeX kernel; if some package uses low-level \vbox es without adding this precaution the \TeX optimization kicks in and no special \par code is executed.

And there is another optimization that is painful: if a paragraph is interrupted by a mathematical display, e.g., $\left[\dots \right]$ in \LaTeX or $\$ \dots \$$ in plain \TeX , then \TeX will resume horizontal mode afterward, i.e., it will start to build a new horizontal list without inserting an indentation box or \everypar at that point. However, if that list immediately ends with an explicit or implicit \par then \TeX will simply throw away this “null” paragraph and not do its usual “end of horizontal list” processing, so this special case also needs to be accounted for when introducing any extended processing.

4.2 The new mechanism implemented for \LaTeX

To improve the situation (and also to support automatic tagging of PDF documents) we now offer public as well as private hooks at the start and end of the paragraph processing. The public hooks can be used by packages (or by the user in the preamble or within the document) and using the hook mechanisms it is possible to reorder or arrange code from different packages in such a way that these can safely coexist.

To make that happen we have to make use of the basic functionality that is offered by \TeX , e.g., we install special code inside \everypar to provide hooks at the beginning and we redefine \par to do some special processing when appropriate to install hooks at the end of the paragraph.

In order to make this work, we have to ensure that package use of \everypar is not overwriting our code. This is done through a trick: we basically hide the real \everypar from the packages and offer them a new token register (with the same name). So if they install their own code it doesn’t overwrite ours. Our code then inserts the new \everypar at the right place inside the process so that it looks as if it was the primitive \everypar .²²

At the end of the paragraph it would be great if we could use a similar trick. However, due to the fact that \TeX inserts the token \par (that doesn’t have a defined meaning) we can’t hide “the real thingTM” and offer the package an indistinguishable alternate.

Fortunately, \LaTeX has already redefined \par for its own purposes. As a result there aren’t many packages that attempt to change \par , because without a lot of extra care that would fail miserably. But the bottom line is that, if you load a package that alters \par then the end of paragraph hooks are most likely not executing while that redefinition is active.²³

²²Ideally, \everypar wouldn’t be used at all by packages and instead they would simply write their code into the hooks now offered by the kernel. However, while this is the longterm goal and clearly an improvement (because then the packages do no longer need to worry about getting their code overwritten or needing to account for already existing code in \everypar), this will not happen overnight. For that reason support for this legacy method is retained.

²³Similarly to the \everypar situation, the remedy is that such packages stop doing this and instead add their alterations into the paragraph hooks now provided.

4.2.1 The provided hooks

`para/before` The following four public hooks are defined and executed for each paragraph:
`para/begin`
`para/end` **para/before** This hook is executed after the kernel hook `\@kernel@before@para@before`
`para/after` (discussed below) in vertical mode immediately after \TeX has contributed `\parskip` to the vertical list and before the actual paragraph processing in horizontal mode starts.

This hook should either not produce any typeset material or add only vertical material. If it starts a paragraph an error is generated. The reason is that we are in the starting process of processing a paragraph and so this would lead to endless recursion.²⁴

para/begin This hook is executed after the kernel hook `\@kernel@before@para@begin` (discussed below) in horizontal mode immediately before the indentation box is placed (if there is any, i.e., if the paragraph hasn't been started with `\noindent`).

The indentation box to be typeset is available to the hook as `\IndentBox` and its automatic placement (after the hook is executed) can be prevented through `\OmitIndent`. More precisely `\OmitIndent` voids the box.

The indentation box is then typeset directly after the hook execution by something equivalent to `\box\IndentBox` followed by the current content of the token register `\everypar` that it is available to the kernel or to packages (that run some legacy code).

One has to be careful not to add any code to the hook that starts its own paragraph (e.g., by adding a `\parbox` or a `\marginpar` inside) because that would call the hook inside again (as a new paragraph is started there) and thus lead to an endless recursion ending only after exhausting the available memory. This can only be done by making sure that is not executed for the inner paragraphs (or at least not recursively forever).

para/end This hook is executed at the end of a paragraph when \TeX is ready to return to vertical mode and after it has removed the last horizontal glue (but not any kerns) placed on the horizontal list. The code is still executed in horizontal mode so it is possible to add further horizontal material at this point, but it should not alter the mode (even a temporary exit from horizontal mode would create chaos—any attempt will cause an error message)! After the hook has ended the kernel hook `\@kernel@after@para@end` is executed and then \TeX returns to vertical mode.

The hook is offered as public hook, but because of the requirement to stay within horizontal mode one needs to be careful in what is placed into the hook.²⁵

This hook is implemented as a reversed hook.

para/after This hook is executed directly after \TeX has returned to vertical mode and after any material that migrated out of the horizontal list (e.g., from a `\vadjust`) has processed.

²⁴One could allow it but only if the newly started paragraph is processed without any hooks. Further more correct spacing would be a bit of a nightmare so for now this is forbidden.

²⁵Maybe we should guard against that, but it would be rather tricky to implement as mode changes can happen across group boundaries so one would need to keep a private stack just for that. Well, something to ponder.

This hook should either not produce any typeset material or add only vertical material. However, for this hook starting a new paragraph is not a disaster so that it isn't prevented.

This hook is implemented as a reversed hook.

Once that hook code has been processed the kernel hook `\@kernel@after@para@after` is executed as the final action of the paragraph processing.

```
\@kernel@before@para@before
\@kernel@after@para@after
\@kernel@before@para@begin
\@kernel@after@para@end
```

As already mentioned above there are also four kernel hooks that are executed at the start and end of the processing.

`\@kernel@before@para@before` For future extensions, not currently used by the kernel.

`\@kernel@after@para@after` For future extensions, not currently used by the kernel.

`\@kernel@before@para@begin` Used by the kernel to implement tagging. This hook is executed at the very beginning of a paragraph after `TEX` has switched to horizontal mode but before any indentation box got added or any `\everypar` was run.

It should not generate typeset material that could alter the position. Note that it should never leave hmode, otherwise you will end with a loop! We could guard against this, but since it is an internal kernel hook that shouldn't be touched this isn't checked.

`\@kernel@after@para@end` Used by the kernel to implement tagging. It is executed directly after the public `para/end` hook. After it there is a quick check that we are still in horizontal mode, i.e., that the public hook has not mistakenly ended horizontal mode prematurely (this is an incomplete check just testing the mode and could perhaps be improved (at the cost of speed)).

4.2.2 Altered and newly provided commands

```
\par
\endgraf
\para_end:
```

An explicit request for ending a paragraph is provided in plain `TEX` under the name `\endgraf`, which simply uses the primitive meaning (regardless of what `\par` may have as its current definition). In `LATEX` `\endgraf` (with that behavior) was originally also available.

With the new paragraph handling in `LATEX`, ending a paragraph means a bit more than just calling the engine's paragraph builder: the process also has to add any hook code for the end of a paragraph. Thus `\endgraf` was changed to provide this additional functionality (along with `\par` remaining subject to its current meaning).

The `expl3` name for this functionality is `\para_end:`.

Note: *The next two commands are still under discussion and may slightly change their semantics (as described in the document) and/or their names between now and the 2021 Spring release!*

<code>\OmitIndent</code>	Inside the <code>para/begin</code> hook one can use this command to suppress the indentation box at the start of the paragraph. (Technically it is possible to use this command outside the hook as well, but this should not be relied upon.) The box itself remains available for use.
<code>\para_omit_indent:</code>	

The `expl3` name for the function is `\para_omit_indent:`.

<code>\IndentBox</code>	The box register holding the indentation box for the paragraph is available for inspection (or changes) inside hooks. It remains available even if the <code>\OmitIndent</code> command was used; in that case it will just not be automatically placed.
<code>\g_para_indent_box</code>	

The `expl3` name for the box register is `\g_para_indent_box`.

<code>\RawIndent</code>	<code>\RawIndent</code> <i>hmode material</i> <code>\RawParEnd</code>
<code>\para_raw_indent:</code>	<code>\RawNoindent</code> <i>hmode material</i> <code>\RawParEnd</code>
<code>\RawNoindent</code>	The commands <code>\RawIndent</code> and <code>\RawNoindent</code> are not meant for normal paragraph building (where the result is a textual paragraph in the traditional meaning of the word), but for special cases where <code>T_EX</code> 's low-level algorithm is used to achieve special effects, but where the result is not a "paragraph".
<code>\para_raw_noindent:</code>	
<code>\RawParEnd</code>	
<code>\para_raw_end:</code>	

They are called "raw", because they bypass `LATEX`'s hook mechanism for paragraphs and simply invoke the low-level `TEX` algorithm. I.e., they are like the original `TEX` primitives `\indent` and `\noindent` (that is they execute no hooks other than `\everypar`) except that they can only be used in vertical mode and generate an error if found elsewhere.

To avoid issues a paragraph started by them should always be ended by `\RawParEnd`²⁶ and not by `\par` (or a blank line), because the latter will execute hooks which then have no counterpart at the beginning of the paragraph. It is the responsibility of the programmer to make sure that they are properly paired. This also means that one should not put arbitrary user content between these commands if that content could contain stray `\pars`.

The `expl3` names for the functions are `\para_raw_indent:`, `\para_raw_indent:` and `\para_raw_end:`.

4.2.3 Examples

None of the examples in this section are meant for real use as they are far too simple-minded but they should give some ideas of what could be possible if a bit more care is applied.

Testing the mechanism

The idea is to output for each paragraph encountered some information: a paragraph sequence number, a level number in roman numerals, the environment in which this paragraph appears, and the line number where the start or end of the paragraph is, e.g., something like

²⁶Technical note for those who know their *T_EXbook*: the `\RawParEnd` command invokes the original `TEX` engine definition of `\par` that (solely) triggers the paragraph builder in `TEX` when found inside unrestricted horizontal mode and does nothing in other processing modes.

```

PARA: 1-i start (document env. on input line 38)
PARA: 1-i end   (document env. on input line 38)
PARA: 2-i start (document env. on input line 40)
PARA: 3-ii start (minipage env. on input line 40)
PARA: 3-ii end   (minipage env. on input line 40)
PARA: 2-i end   (document env. on input line 41)

```

As you can see paragraph 2 starts on line 40 and ends on 41 and inside a minipage started paragraph 3 (start and end on line 40). If you run this on some document you will find that L^AT_EX considers more things “a paragraph” than you have probably thought.

This was generated by the following hook code:

```

\newcounter{paracnt}          % sequence counter
\newcounter{paralevel}       % level counter

```

To support paragraph nesting we need to maintain a stack of the sequence numbers. This is most easily done using `expl3` functions, so we switch over. This is not a very general implementation, just enough for what we need and a bit of L^AT_EX 2_ε thrown in as well. When popping, the result gets stored in `\paracntvalue` and the `\ERROR` should never happen because it means we have tried to pop from an empty stack.

```

\ExplSyntaxOn
\seq_new:N \g_para_seq
\cs_new:Npn \ParaPush
  {\seq_gpush:No \g_para_seq {\the\value{paracnt}}}
\cs_new:Npn \ParaPop  {\seq_gpop:MMF \g_para_seq \paracntvalue \ERROR }
\ExplSyntaxOff

```

At the start of the paragraph increment both sequence counter and level and also save the then current sequence number on our stack.

```

\makeatletter % because we use a few internal Te commands
\AddToHook{para/begin}{%
  \stepcounter{paracnt}\stepcounter{paralevel}%
  \ParaPush
}

```

To display the sequence number we `\typeout` the current sequence and level number. The command `\@currenenv` gives us the current environment and `\on@line` produces a space and the current input line number.

```

\typeout{PARA: \arabic{paracnt}-\roman{paralevel} start
  (\@currenenv\space env.\on@line)}%

```

We also typeset the sequence number as a tiny red number in a box that takes up no horizontal space. This helps us seeing where L^AT_EX sees the start and end of the paragraphs in the document.

```

\llap{\color{red}\tiny\arabic{paracnt}\ }%
}

```

At the end of the paragraph we display sequence number and level again. The level counter has the correct value but we need to retrieve the right sequence value by popping it off the stack after which it is available in `\paracntvalue` the way we have set this up above.

```

\AddToHook{para/end}{%
  \ParaPop
  \typeout{PARA: \paracntvalue-\roman{paralevel} end \space\space
    (\@currenvir\space env.\on@line)}%
}

```

We also typeset again a tiny red number with that value, this time sticking out to the right.²⁷ We also decrement the level counter since our level has finished.

```

  \rlap{\color{red}\tiny\ \paracntvalue}%
  \addtocounter{paralevel}{-1}%
}
\makeatother

```

Mark the first paragraph of each itemize

The code for this is rather simple. We supply some code that is executed only once inside a hook at the start of each `itemize`. We explicitly change the color back and forth so that we don't introduce grouping around the paragraph.

```

\AddToHook{env/itemize/begin}{%
  \AddToHookNext{para/begin}{\color{blue}}%
  \AddToHookNext{para/end}{\color{black}}%
}

```

As a result the first paragraph of each `itemize` will appear in blue.

4.2.4 Some technical notes

The code tries hard to be transparent for package code, but of course any change means that there is a potential for breaking other code. So in section we collect a few cases that may be of importance if low-level code is dealing with paragraphs that are now behaving slightly differently. The notes are from issues we observed and will probably grow over time.

Glue items between paragraphs (found with `fancypar`)

In the past \LaTeX placed two glue items between two consecutive paragraphs, e.g.,

```
text1 \par text2 \par
```

would show something like

```

\glue(\parskip) 0.0 plus 1.0
\glue(\baselineskip) 5.16669

```

but now there is another `\parskip` glue (that is always 0pt):

```

\glue(\parskip) 0.0 plus 1.0
\glue(\parskip) 0.0
\glue(\baselineskip) 5.16669

```

²⁷Note that this can alter the document pagination, because a paragraph ending in a display (e.g., an equation) will get an extra line—in that case our tiny number has an effect even though it doesn't take up any space, because it paragraph is no longer empty and thus isn't dropped!

The reason is that we generate a “fake” paragraph to gain control and safely add the early hooks, but this generates an additional glue item. That item doesn’t contribute anything vertically but if somebody writes code that unravels a constructed list using `\lastbox`, `\unskip` and `\unpenalty` then the code has to remove one additional glue item or else it will fail.

Chapter 5

The shipout routine: hooks and interfaces

5.1 Introduction

The code provides an interface to the `\shipout` primitive of \TeX which is called when a finished page is finally “shipped out” to the target output file, e.g., the `.dvi` or `.pdf` file. A good portion of the code is based on ideas by Heiko Oberdiek implemented in his packages `atbegshi` and `atenddvi` even though the interfaces are somewhat different.²⁸

5.1.1 Overloading the `\shipout` primitive

`\shipout` With this implementation \TeX 's `shipout` primitive is no longer available for direct use. Instead `\shipout` is running some (complicated) code that picks up the box to be shipped out regardless of how that is done, i.e., as a constructed `\vbox` or `\hbox` or as a box register.

It then stores it in a named box register. This box can then be manipulated through a set of hooks after which it is shipped out for real.

Each `shipout` that actually happens (i.e., where the material is not discarded for one or the other reason) is recorded and the total number is available in a readonly variable and in a \LaTeX counter.

²⁸Heiko's interfaces are emulated by the kernel code, if a document requests his packages, so older documents will continue to work.

`\RawShipout` This command implements a simplified shipout that bypasses the foreground and background hooks, e.g., only `shipout/firstpage` and `shipout/lastpage` are executed and the total shipout counters are incremented.

The command doesn't use `\ShipoutBox` but its own private box register so that it can be used inside of shipout hooks to do some additional shipouts while already in the output routine with the current page being stored in `\ShipoutBox`. It does have access to `\ShipoutBox` if it is used in `shipout/before` (or `shipout/after`) and can use its content.

It is safe to use it in `shipout/before` or `shipout/after` but not necessarily in the other `shipout/...` hooks as they are intended for special processing.

`\ShipoutBox`
`\l_shipout_box` This box register is called `\ShipoutBox` (alternatively available via the L3 name `\l_shipout_box`).

This box is a “local” box and assignments to it should be done only locally. Global assignments (as done by some packages with older code where this box is known as 255) may work but they are conceptually wrong and may result in errors under certain circumstances.

During the execution of `shipout/before` this box contains the accumulated material for the page, but not yet any material added by other shipout hooks. During execution of `shipout/after`, i.e., after the shipout has happened, the box also contains any background or foreground material.

Material from the hooks `shipout/firstpage` or `shipout/lastpage` is not included (but only used during the actual shipout) to facilitate reuse of the box data (e.g., `shipout/firstpage` material should never be added to a later page of the output).

`\l_shipout_box_ht_dim`
`\l_shipout_box_dp_dim`
`\l_shipout_box_wd_dim`
`\l_shipout_box_ht_plus_dp_dim`

The shipout box dimensions are available in the L3 registers `\l_shipout_box_ht_dim`, etc. (there are no $\text{\LaTeX} 2_{\epsilon}$ names).²⁹ These variables can be used inside the hook code for `shipout/before`, `shipout/foreground` and `shipout/background` if needed.

²⁹Might need changing, but HO's version as strings is not really helpful I think).

5.1.2 Provided hooks

<code>shipout/before</code>	The code for <code>\shipout</code> offers a number of hooks into which packages (or the user) can add code to support different use cases. These are:
<code>shipout/after</code>	
<code>shipout/foreground</code>	
<code>shipout/background</code>	
<code>shipout/firstpage</code>	
<code>shipout/lastpage</code>	

shipout/before This hook is executed after the finished page has been stored in `\ShipoutBox` / `\l_shipout_box`. It can be used to alter that box content or to discard it completely (see `\DiscardShipoutBox` below).

You can use `\RawShipout` inside this hook for special use cases. It can make use of `\ShipoutBox` (which doesn't yet include the background and foreground material).

Note: It is not possible (or say advisable) to try and use this hook to typeset material with the intention to return it to main vertical list, it will go wrong and give unexpected results in many cases—for starters it will appear after the current page not before or it will vanish or the vertical spacing will be wrong!

shipout/background This hook adds a picture environment into the background of the page with the (0,0) coordinate in the top-left corner using a `\unitlength` of 1pt.

It should therefore only receive `\put` commands or other commands suitable in a `picture` environment and the vertical coordinate values would normally be negative.

Technically this is implemented by adding a zero-sized `\hbox` as the very first item into the `\ShipoutBox` containing that `picture` environment. Thus the rest of the box content will overprint what ever is typeset by that hook.

shipout/foreground This hook adds a picture environment into the foreground of the page with the (0,0) coordinate in the top-left corner using a `\unitlength` of 1pt.

Technically this is implemented by adding a zero-sized `\hbox` as the very last item into the `\ShipoutBox` and raising it up so that it still has its (0,0) point in the top-left corner. But being placed after the main box content it will be typeset later and thus overprints it (i.e., is in the foreground).

shipout This hook is executed after foreground and/or background material has been added, i.e., just in front of the actual shipout operation. Its purpose is to allow manipulation of the finalized box (stored in `\ShipoutBox`) with the extra material also in place (which is not yet the case in `shipout/before`).

It cannot be used to cancel the shipout operation via `\DiscardShipoutBox` (that has to happen in `shipout/before`, if desired!

shipout/firstpage The material from this hook is executed only once at the very beginning of the first output page that is shipped out (i.e., not discarded at the last minute). It should only contain `\special` or similar commands needed to direct post processors handling the `.dvi` or `.pdf` output.³⁰

This hook is added to the very first page regardless of how it is shipped out (i.e., with `\shipout` or `\RawShipout`).

³⁰In $\text{\LaTeX}2_{\epsilon}$ that was already existing, but implemented using a box register with the name `\@begindivibox`.

shipout/lastpage The corresponding hook to add `\specials` at the very end of the output file. It is only executed on the very last page of the output file — or rather on the page that \LaTeX believes is the last one. Again it is executed regardless of the shipout method.

It may not be possible for \LaTeX to correctly determine which page is the last one without several reruns. If this happens and the hook is non-empty then \LaTeX will add an extra page to place the material and also request a rerun to get the correct placement sorted out.

shipout/after This hook is executed after a shipout has happened. If the shipout box is discarded this hook is not looked at.

You can use `\RawShipout` inside this hook for special use cases and the main `\ShipoutBox` is still available at this point (but in contrast to `shipout/before` it now includes the background and foreground material).

Note: Just like `shipout/before` this hook is not meant to be used for adding typeset material back to the main vertical list—it might vanish or the vertical spacing will be wrong!

As mentioned above the hook `shipout/before` is executed first and can manipulate the prepared shipout box stored in `\ShipoutBox` or set things up for use in `\write` during the actual shipout. It is even run if there was a `\DiscardShipoutBox` request in the document.

The other hooks (except `shipout` and `shipout/after`) are added inside hboxes to the box being shipped out in the following order:

<code>shipout/firstpage</code>	only on the first page
<code>shipout/background</code>	
<code>\boxed content of \ShipoutBox</code>	
<code>shipout/foreground</code>	
<code>shipout/lastpage</code>	only on the last page

If any of the hooks has no code then the corresponding box is added at that point.

Once the (page) box has got the above extra content it can again be manipulated using the `shipout` hook and then is shipped out for real.

Once the (page) box has been shipped out the `shipout/after` hook is called (while you are still inside the output routine). It is not called if the shipout box was discarded.

In a document that doesn't produce pages, e.g., only makes `\typeouts`, none of the hooks are ever executed (as there is no `\shipout`) not even the `shipout/lastpage` hook.

If `\RawShipout` is used instead of `\shipout` then only the hooks `shipout/firstpage` and `shipout/lastpage` are executed (on the first or last page), all others are bypassed.

5.1.3 Legacy \LaTeX commands

`\AtBeginDvi` `\AtBeginDvi` `{(code)}`

`\AtEndDvi` `\AtBeginDvi` is the existing $\text{\LaTeX} 2_{\epsilon}$ interface to fill the `shipout/firstpage` hook. This is not really a good name as it is not just supporting `.dvi` but also `.pdf` output or `.xdv`.

`\AtEndDvi` is the counterpart that was not available in the kernel but only through the package `atenddvi`. It fills the `shipout/lastpage` hook.

Neither interface can set a code label but uses the current default label.

As these two wrappers have been available for a long time we continue offering them (but not enhancing them, e.g., by providing support for code labels).

For new code we strongly suggest using the high-level hook management commands directly instead of “randomly-named” wrappers. This will lead to code that is easier to understand and to maintain and it also allows you to set code labels if needed.

For this reason we do not provide any other “new” wrapper commands for the above hooks in the kernel, but only keep the existing ones for backward compatibility.

5.1.4 Special commands for use inside the hooks

<code>\DiscardShipoutBox</code>	<code>\AddToHookNext {shipout/before} {...\DiscardShipoutBox...}</code>
<code>\shipout_discard:</code>	

The `\DiscardShipoutBox` declaration (L3 name `\shipout_discard:`) requests that on the next shipout the page box is thrown away instead of being shipped to the `.dvi` or `.pdf` file.

Typical applications wouldn't do this unconditionally, but have some processing logic that decides to use or not to use the page.

Note that if this declaration is used directly in the document it may depend on the placement to which page it applies, given that \LaTeX output routine is called in an asynchronous manner! Thus normally one would use this only as part of the `shipout/before` code.

Todo: Once we have a new mark mechanism available we can improve on that and make sure that the declaration applies to the page that contains it — not done (yet)

`\DiscardShipoutBox` cannot be used in any of the `shipout/...` hooks other than `shipout/before`.

In the `atbegshi` package there are a number of additional commands for use inside the `shipout/before` hook. They should normally not be needed any more as one can instead simply add code to the hooks `shipout/before`, `shipout`, `shipout/background` or `shipout/foreground`.³¹ If `atbegshi` gets loaded then those commands become available as public functions with their original names as given below.

5.1.5 Provided Lua \TeX callbacks

<code>pre_shipout_filter</code>	Under Lua \TeX the <code>pre_shipout_filter</code> Lua callback is provided which gets called directly after the <code>shipout</code> hook, immediately before the <code>shipout</code> primitive gets invoked. The signature is
---------------------------------	--

```
function(<node> head)
  return true
end
```

The `head` is the list node corresponding to the box to be shipped out. The return value should always be `true`.

³¹If that assumption turns out to be wrong it would be trivial to change them to public functions (right now they are private).

5.1.6 Information counters

<code>\ReadonlyShipoutCounter</code>	<code>\ifnum\ReadonlyShipoutCounter=...</code>
<code>\g_shipout_readonly_int</code>	<code>\int_use:N \g_shipout_readonly_int % expl3 usage</code>

This integer holds the number of pages shipped out up to now (including the one to be shipped out when inside the output routine). More precisely, it is incremented only after it is clear that a page will be shipped out, i.e., after the `shipout/before` hook (because that might discard the page)! In contrast `shipout/after` sees the incremented value.

Just like with the `page` counter its value is only accurate within the output routine. In the body of the document it may be off by one as the output routine is called asynchronously!

Also important: it *must not* be set, only read. There are no provisions to prevent that restriction, but if you manipulate it, chaos will be the result. To emphasize this fact it is not provided as a \LaTeX counter but as a \TeX counter (i.e., a command), so `\Alph{\ReadonlyShipoutCounter}` etc, would not work.

<code>totalpages</code>	<code>\arabic{totalpages}</code>
<code>\g_shipout_totalpages_int</code>	<code>\int_use:N \g_shipout_totalpage_int % expl3 usage</code>

In contrast to `\ReadonlyShipoutCounter`, the `totalpages` counter is a \LaTeX counter and incremented for each shipout attempt including those pages that are discarded for one or the other reason. Again `shipout/before` sees the counter before it is incremented. In contrast `shipout/after` sees the incremented value.

Furthermore, while it is incremented for each page, its value is never used by \LaTeX . It can therefore be freely reset or changed by user code, for example, to additionally count a number of pages that are not build by \LaTeX but are added in a later part of the process, e.g., cover pages or picture pages made externally.

Important: as this is a page-related counter its value is only reliable inside the output routine!

<code>\PreviousTotalPages</code>	<code>\PreviousTotalPages</code>
----------------------------------	----------------------------------

Command that expands to the number of total pages from the previous run. If there was no previous run or if used in the preamble it expands to 0. Note that this is a command and not a counter, so in order to display the number in, say, Roman numerals you have to assign its value to a counter and then use `\Roman` on that counter.

5.1.7 Debugging shipout code

<code>\DebugShipoutsOn</code>	<code>\DebugShipoutsOn</code>
<code>\DebugShipoutsOff</code>	Turn the debugging of shipout code on or off. This displays changes made to the shipout data structures.
<code>\shipout_debug_on:</code>	
<code>\shipout_debug_off:</code>	

Todo: This needs some rationalizing and may not stay this way.

5.2 Emulating commands from other packages

The packages in this section are no longer necessary, but as they are used by other packages, they are emulated when they are explicitly loaded with `\usepackage` or `\RequirePackage`.

Please note that the emulation only happens if the package is explicitly requested, i.e., the commands documented below are not automatically available in the L^AT_EX kernel! If you write a new package we suggest to use the appropriate kernel hooks directly instead of loading the emulation.

5.2.1 Emulating `atbegshi`

<code>\AtBeginShipoutUpperLeft</code>	<code>\AddToHook {shipout/before} {...\AtBeginShipoutUpperLeft{<code>}...}</code>
<code>\AtBeginShipoutUpperLeftForeground</code>	

This adds a `picture` environment into the background of the shipout box expecting `<code>` to contain `picture` commands. The same effect can be obtained by simply using kernel features as follows:

```
\AddToHook{shipout/background}{<code>}
```

There is one technical difference: if `\AtBeginShipoutUpperLeft` is used several times each invocation is put into its own box inside the shipout box whereas all `<code>` going into `shipout/background` ends up all in the same box in the order it is added or sorted based on the rules for the hook chunks.

`\AtBeginShipoutUpperLeftForeground` is similar with the difference that the `picture` environment is placed in the foreground. To model it with the kernel functions use the hook `shipout/foreground` instead.

<code>\AtBeginShipoutAddToBox</code>	<code>\AddToHook {shipout/before} {...\AtBeginShipoutAddToBox{<code>}...}</code>
<code>\AtBeginShipoutAddToBoxForeground</code>	

These work like `\AtBeginShipoutUpperLeft` and `\AtBeginShipoutUpperLeftForeground` with the difference that `<code>` is directly placed into an `\hbox` inside the shipout box and not surrounded by a `picture` environment.

To emulate them using `shipout/background` or `shipout/foreground` you may have to wrap `<code>` into a `\put` statement but if the code is not doing any typesetting just adding it to the hook should be sufficient.

<code>\AtBeginShipoutBox</code>	This is the name of the shipout box as <code>atbegshi</code> knows it.
---------------------------------	--

<code>\AtBeginShipoutOriginalShipout</code>

This is the name of the `\shipout` primitive as `atbegshi` knows it. This bypasses all the mechanisms set up by the L^AT_EX kernel and there are various scenarios in which it can therefore fail. It should only be used to run existing legacy `atbegshi` code but not in newly developed applications.

The kernel alternative is `\RawShipout` which is integrated with the L^AT_EX mechanisms and updates, for example, the `\ReadonlyShipoutCounter` counter. Please use `\RawShipout` for new code if you want to bypass the before, foreground and background hooks.

`\AtBeginShipoutInit` By default `atbegshi` delayed its action until `\begin{document}`. This command was forcing it in an earlier place. With the new concept it does nothing.

`\AtBeginShipout` `\AtBeginShipout{<code>} ≡ \AddToHook{shipout/before}{<code>}`
`\AtBeginShipoutNext` `\AtBeginShipoutNext{<code>} ≡ \AddToHookNext{shipout/before}{<code>}`

This is equivalent to filling the `shipout/before` hook by either using `\AddToHook` or `\AddToHookNext`, respectively.

`\AtBeginShipoutFirst` The `atbegshi` names for `\AtBeginDvi` and `\DiscardShipoutBox`.
`\AtBeginShipoutDiscard`

5.2.2 Emulating `everyshi`

The `everyshi` package is providing commands to run arbitrary code just before the shipout starts. One point of difference: in the new shipout hooks the page is available as `\ShipoutBox` for inspection of change, one should not manipulate box 255 directly inside `shipout/before`, so old code doing this would change to use `\ShipoutBox` instead of 255 or `\@cclv`.

`\EveryShipout` `\EveryShipout{<code>} ≡ \AddToHook{shipout/before}{<code>}`

`\AtNextShipout` `\AtNextShipout{<code>} ≡ \AddToHookNext{shipout/before}{<code>}`

However, most use cases for `everyshi` are attempts to put some picture or text into the background or foreground of the page and that can be done today simply by using the `shipout/background` and `shipout/foreground` hooks without any need to coding.

5.2.3 Emulating `atenddvi`

The `atenddvi` package implemented only a single command: `\AtEndDvi` and that is now available out of the box so the emulation makes the package a no-op.

5.2.4 Emulating `everypage`

This package patched the original `\@begindvi` hook and replaced it with its own version. Its functionality is now covered by the hooks offered by the kernel so that there is no need for such patching any longer.

`\AddEverypageHook` `\AddEverypageHook{<code>} ≡`
`\AddToHook{shipout/background}{\put(1in,-1in){<code>}}`

`\AddEverypageHook` is adding something into the background of every page at a position of 1in to the right and 1in down from the top left corner of the page. By using the kernel hook directly you can put your material directly to the right place, i.e., use other coordinates in the `\put` statement above.

`\AddThispageHook` `\AddThispageHook{<code>} ≡`
`\AddToHookNext{shipout/background}{\put(1in,-1in){<code>}}`

The `\AddThispageHook` wrapper is similar but uses `\AddToHookNext`.

Part II
Run data and page design

Chapter 6

The marks mechanism

6.1 Introduction

The $\text{T}_{\text{E}}\text{X}$ engines offer a low-level mark mechanism to communicate information about the content of the current page to the asynchronous operating output routine. It works by placing `\mark` commands into the source document. When the material for the current page is assembled in box 255, $\text{T}_{\text{E}}\text{X}$ scans for such marks and sets the commands `\topmark`, `\firstmark` and `\botmark`. The `\firstmark` receives the content of the first `\mark` seen in box 255 and `\botmark` the content of the last mark seen. The `\topmark` holds the content of the last mark seen on the previous page or more exactly the value of `\botmark` from the previous page. If there are no marks on the current page then all three are made equal to the `\botmark` from the previous page.

This mechanism works well for simple formats (such as plain $\text{T}_{\text{E}}\text{X}$) whose output routines are only called to generate pages. It fails, however, in $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ (and other more complex formats), because here the output routine is sometimes called without producing a page, e.g., when encountering a float and placing it into one of the float regions. In that case the output routine is called, determines where to place the float, alters the goal for assembling text material (if the float was added to the top or bottom region) and then it resumes collecting textual material.

As a result the `\botmark` gets updated and so `\topmark` no longer reflects the situation at the top of the next page when that page is finally boxed.

Another problem for $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ was that it wanted to use several “independent” marks and in the early implementations of $\text{T}_{\text{E}}\text{X}$ there was only a single `\mark` command available. For that reason $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ implemented its own mark mechanism where the marks always contained two parts with their own interfaces: `\markboth` and `\markright` to set marks and `\leftmark` and `\rightmark` to retrieve them.

However, this extended mechanism (while supporting scenarios such as chapter/section marks) was far from general. The mark situation at the top of a page (i.e., `\topmark`) remained unusable and the two marks offered were not really independent of each other because `\markboth` (as the name indicates) was always setting both.

The new mechanism overcomes both issues:

- It provides arbitrarily many, fully independent named marks, that can be allocated and, from that point onwards, used.
- It offers access for each such marks to retrieve its top, first, and bottom values separately.

- Furthermore, the mechanism is augmented to give access to marks in different “regions” which may not be just full pages.

6.2 Design-level and code-level interfaces

The interfaces are mainly meant for package developers, but they are usable (with appropriate care) also in the document preamble, for example, when setting up special running headers with `fancyhdr`, etc. They are therefore available both as CamelCase commands as well as commands for use in the L3 programming layer. Both are described together below.

<code>\NewMarkClass</code>	<code>\NewMarkClass {<class>}</code>
<code>\mark_new_class:n</code>	<code>\mark_new_class:n {<class>}</code>

Declares a new `<class>` of marks to be tracked by L^AT_EX. Each `<class>` must be declared before it is used.

Mark classes can only be declared before `\begin{document}`.

<code>\InsertMark</code>	<code>\InsertMark {<class>} {<text>}</code>
<code>\mark_insert:nn</code>	<code>\mark_insert:nn {<class>} {<text>}</code>

Adds a mark to the current galley for the `<class>`, containing the `<text>`.

It has no effect in places in which you can't place floats, e.g., a mark inside a box or inside a footnote never shows up anywhere.

If used in vertical mode it obeys L^AT_EX's internal `@nobreak` switch, i.e., it does not introduce a breakpoint if used after a heading. If used in horizontal mode it doesn't handle spacing (like, for example, `\index` or `\label` does, so it should be attached to material that is typeset.

<code>insertmark</code>	<code>\AddToHook {insertmark} {<code>}</code>
-------------------------	---

When marks are inserted, the mark content may need some special treatment, e.g., by default `\label`, `\index`, and `\glossary` do not expand at this time (but only later if and when the mark content is actually used. In order to allow packages to augment or alter this setup there is a public hook `insertmark` that is executed at this point. It runs in a group so local modification to commands are only applied to the `<text>` argument of `\InsertMark` or `\mark_insert:nn`.

<code>\TopMark</code>	<code>*</code>	<code>\TopMark</code>	<code>[(<i>region</i>)]</code>	<code>{(<i>class</i>)}</code>
<code>\FirstMark</code>	<code>*</code>	<code>\FirstMark</code>	<code>[(<i>region</i>)]</code>	<code>{(<i>class</i>)}</code>
<code>\LastMark</code>	<code>*</code>	<code>\LastMark</code>	<code>[(<i>region</i>)]</code>	<code>{(<i>class</i>)}</code>
<code>\mark_use_top:nn</code>	<code>*</code>	<code>\mark_use_top:nn</code>	<code>{(<i>region</i>)}</code>	<code>{(<i>class</i>)}</code>
<code>\mark_use_first:nn</code>	<code>*</code>	<code>\mark_use_first:nn</code>	<code>{(<i>region</i>)}</code>	<code>{(<i>class</i>)}</code>
<code>\mark_use_last:nn</code>	<code>*</code>	<code>\mark_use_last:nn</code>	<code>{(<i>region</i>)}</code>	<code>{(<i>class</i>)}</code>

These functions expand to the appropriate mark `<text>` for the given `<class>` in the specified `<region>`. The default `<region>` in the design-level commands is `page`. Note that with the L3 layer commands there are no optional arguments, i.e., both arguments have to be provided.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<text>` does not expand further when appearing in an `x`-type or `e`-type argument expansion.

The “first” and “last” marks are those seen first and last in the current region/page, respectively. The “top” mark is the last mark of the `<class>` seen in an earlier region, i.e., the `<text>` what would be “current” at the very top of the region.

Important!

The commands are only meaningful inside the output routine, in other places their result is (while not random) unpredictable due to the way L^AT_EX cuts text material into pages. There is, however, one exception: if you produce multiple columns using the `multicol` package, it is possible to retrieve mark values from the regions `first-column`, `last-column`, `mcol-1`, `mcol-2`,... directly after the environment has ended. This can, for example, be useful if a `multicols` has been used inside a box.

Currently, `<region>` is one of `page`, `previous-page`, `column`, `previous-column`, `first-column`, `last-column`, and `mcol-1` (first column in a `multicols`), `mcol-2` (second column in a `multicols`), up to `mcol-20` (twentieth column in a `multicols`). See section 6.2.2 for discussion of how these regions behave and how one can make use of them.

<code>\IfMarksEqualTF</code>	<code>*</code>	<code>\IfMarksEqualTF</code>	<code>[(<i>region</i>)]</code>	<code>{(<i>class</i>)}</code>	<code>{(<i>pos</i>₁)}</code>	<code>{(<i>pos</i>₂)}</code>	<code>{(true)}</code>	<code>{(false)}</code>
<code>\IfMarksEqualT</code>	<code>*</code>	<code>\mark_if_eq:nnnnTF</code>	<code>{(<i>region</i>)}</code>	<code>{(<i>class</i>)}</code>	<code>{(<i>pos</i>₁)}</code>	<code>{(<i>pos</i>₂)}</code>	<code>{(true)}</code>	<code>{(false)}</code>
<code>\IfMarksEqualF</code>	<code>*</code>	<code>\mark_if_eq:nnnnnnTF</code>	<code>{(<i>region</i>₁)}</code>	<code>{(<i>class</i>₁)}</code>	<code>{(<i>pos</i>₁)}</code>			
<code>\mark_if_eq:nnnnTF</code>	<code>*</code>		<code>{(<i>region</i>₂)}</code>	<code>{(<i>class</i>₂)}</code>	<code>{(<i>pos</i>₂)}</code>	<code>{(true)}</code>	<code>{(false)}</code>	
<code>\mark_if_eq:nnnnnnTF</code>	<code>*</code>	These conditionals allow you to compare the content of two marks and act based on the result. The commands work in an expansion context, if necessary.						

It is quite common when programming with marks to need to interrogate conditions such as whether marks have appeared on a previous page, or if there are multiple marks present on the current page, and so on. The tests above allow for the construction of a variety of typical test scenarios, with three examples presented below.

The first two conditionals cover only the common scenarios. Both marks are picked up from the same `<region>` (by default `page`) and they have to be of the same `<class>`.³² The `<posi>` argument can be either `top`, `first`, or `last`.

Important to note is that the comparison is not with respect to the textual content of the marks but whether or not they originated from the same `\InsertMark` command (or the L3 layer version `\mark_insert:nn`).

If you wish to compare marks across different regions or across different classes, you have to do it using the generic test only available in the L3 programming layer or do it

³²If an undeclared mark class is used the tests return `true` (not an error).

manually, i.e., get the marks and then compare the values yourself.³³

6.2.1 Use cases for conditionals

However, the basic version is enough for the following typical use cases:

Test for at most one mark of class `myclass` on current page: If the first and last mark in a region are the same then either there was no mark at all, or there was at most one. To test this on the current page:

```
\NewMarkClass{myclass}
\IfMarksEqualTF{myclass}{first}{last}
{ <zero or one mark> }{ <two or more marks> }
```

Test for no mark of class `myclass` in the previous page: If the top mark is the same as the first mark, there is no mark in the region at all. If we wanted to do this test for the previous page:

```
\IfMarksEqualTF[previous-page]{myclass}{top}{first}
{ <no marks> }{ <at least one mark> }
```

Comparing `top` and `last` would give you the same result.

Test for zero, one, or more than one: Combining the two tests from above you can test for zero, one or more than one mark.

```
\IfMarksEqualTF{myclass}{top}{first}
{ <no marks> }
{\IfMarksEqualTF{myclass}{first}{last}
{ <exactly one mark> }{ <more than one mark> }}
```

If you need one of such tests more often (or if you want a separate command for it for readability), then consider defining:

```
\providecommand\IfNoMarkTF[2][page]{\IfMarksEqualTF[#1]{#2}{first}{last}}
```

6.2.2 Understanding regions

If a page has just been finished then the region `page` refers to the current page and `previous-page`, as the name indicates, refers to the page before the current page. This means you are able to access mark information for the current page as well as for the page before (as long as you are inside the output routine) without the need to explicitly save that information beforehand. The `page` region is the region that is most often queried, which is why commands like `\FirstMark` use that region by default.

In single column documents the `column` is the same as the `page` region, but in two-column documents (if not produced by `multicols`), `column` refers to the current column that just got finished and `previous-column` to the one previously finished. Code for running headers is (in standard L^AT_EX) evaluated only after both columns have been assembled, which is another way of saying that in that case `previous-column` refers to the left column and `column` to the right column. However, to make these somewhat

³³If two undeclared mark classes are compared the result is always *true*; if a declared and an undeclared mark class is used it is always *false*.

easier to use, there are also aliased names for these two regions: `first-column` and `last-column`.³⁴

Note that you can only look backwards at already processed regions, e.g., in a `twoside` document finishing a recto (odd, right-hand) page you can access the data from the facing verso (left-hand) page, but if you are finishing a left-hand page you can't integrate data from the upcoming right-hand page. If such a scenario needs to be realized then it is necessary to save the left-hand page temporarily instead of finalizing it, process material for the right-hand page and once both are ready, attach running headers and footers and shipout out both in one go.³⁵

The situation starts getting rather complex if you allow for multiple columns in the way they are supported by the `multicol` package. In this case you might have a variable number of "columns" on a single page to be shipped out. And even if not, then a `multicols` might start or end in the middle of the page; in either case, the regions `column` and `previous-column` become rather meaningless and you should therefore not use them.³⁶ Instead, the algorithm offers `mcol-1`, `mcol-2`, `mcol-3`, etc., to represent the columns in the `multicols` on the current page to be shipped out. If there is more than one `multicols` on the current page then in the output routine only the columns of the last one will be accessible.

These provisions cover, out of the box, a number of layouts and use cases, but obviously not all. However, more cases can be supported by storing away mark information during the processing. Here is the full algorithm:

- The `column` region is used by the "current column" that is being built (moving through all columns with `previous-column` trailing behind (to handle top marks properly).
- When the `multicols` starts, the `column` region is cleared, i.e., from that point on it looks as if there have not been any marks so far. This will make sure that the top mark in the first column is always empty.
- If the `multicols` extends beyond the current page, then the material designated for the current page is split into columns. The `column` region is used to represent each column in turn.
 - First we copy the current data from `column` to `previous-column`. Then the mark data from the current column is placed into the `column` region. Then we alias `column` to `mcol-1`.
 - These steps are repeated for all columns of the `multicols` environment.
 - Finally, the first and the last column of that page is also made available as `first-column` and `last-column`, respectively.
- All those marks inside any of the columns are also available in the `page` region. Thus, if you are interested in the top, first, or last mark of a specific class on the whole page you simply need to query for it in the `page` region.

³⁴The region is called "last-column" not "second-column" in anticipation of extending the mechanism to multiple columns, where first and last would still make sense. There aren't any `previous-first-column` and `previous-last-column` regions to access the corresponding columns from the previous page.

³⁵As of now that scenario is not (yet) officially supported but it would be possible to achieve this using the shipout hooks to store the verso page and then on the next shipout use the hook to shipout both with running headers and footers attached.

³⁶They return something, because they represent the last two columns of the `multicols` when you are inside the output routine, but that is obviously of little use.

- If the `multicols` continues across several pages then this algorithm above is repeated for each page, except that the `column` region is not cleared again. This means that the top mark of the first column of the next page will be the last mark of the last column from the previous page.
- When the `multicols` finishes the remaining material for the current page is balanced to produce columns of roughly equal height.
- Again `column` and `previous-column` are used while this balancing happens and `mcol-1`, `mcol-2`, etc., are used to represent the column regions and `first-column` and `last-column` are set appropriately.
- Then the balanced set of columns is returned back to the page (since there may be space for further material). In addition, all marks inside that material are reinserted so that they become available in the `page` region.
- As a side effect, it is possible (and useful in certain circumstances) to query for mark classes directly after the `multicols` has ended without the need to be inside the output routine. The regions that can be queried this way are `mcol-1`, `mcol-2`, etc. (up to the number of columns the multicol had) and `first-column` and `last-column`.

6.2.3 Debugging mark code

<code>\DebugMarksOn</code> <code>\DebugMarksOff</code> <code>\mark_debug_on:</code> <code>\mark_debug_off:</code>	<code>\DebugMarksOn ... \DebugMarksOff</code> <p>Commands to turn the debugging of mark code on or off. The debugging output is rather coarse and not really intended for normal use at this point in time.</p>
--	--

6.3 Application examples

If you want to figure out if a break was taken at a specific point, e.g., whether a heading appears at the top of the page, you can do something like this:

```

\newcounter{breakcounter}
\NewMarkClass{break}
\newcommand\markedbreak[1]{\stepcounter{breakcounter}%
                          \InsertMark{break}{\arabic{breakcounter}}%
                          \penalty #1\relax
                          \InsertMark{break}{-\arabic{breakcounter}}}

```

To test if the break was taken you can test if `\TopMark{break}` is positive (taken) or negative (not taken) or zero (there was never any marked break so far). The absolute value can be used to keep track of which break it was (with some further coding).

to be extended with additional application examples

6.4 Legacy L^AT_EX 2_ε interface

Here we describe the interfaces that L^AT_EX 2_ε offered since the early nineties and some minor extensions.

6.4.1 Legacy design-level and document-level interfaces

```
\markboth \markboth {(left)} {(right)}  
\markright \markright {(right)}
```

L^AT_EX 2_ε uses two marks which aren't fully independent. A “left” mark generated by the first argument of `\markboth` and a “right” mark generated by the second argument of `\markboth` or by the only argument of `\markright`. The command `\markboth` and `\markright` are in turn called from heading commands such as `\chaptermark` or `\sectionmark` and their behavior is controlled by the document class.

For example, in the `article` class with `twoside` in force the `\sectionmark` will issue `\markboth` with an empty second argument and `\subsectionmark` will issue `\markright`. As a result the left mark will contain chapter titles and the right mark subsection titles.

Note, however, that in one-sided documents the standard behavior is that only `\markright` is used, i.e., there will only be right-marks but no left marks!

```
\leftmark * \leftmark  
\rightmark * \rightmark
```

These functions return the appropriate mark value from the current page and work as before, that is `\leftmark` will get the last (!) left mark from the page and `\rightmark` the first (!) right mark.

In other words they work reasonably well if you want to show the section title that is current when you are about to turn the page and also show the first subsection title on the current page (or the last from the previous page if there wasn't one). Other combinations can't be shown using this interface.

The commands are fully expandable, because this is how they have been always defined in L^AT_EX. However, this is of course only true if the content of the mark they return is itself expandable and does not contain any fragile material. Given that this can't be guaranteed for arbitrary content, a programmer using them in this way should use `\protected@edef` and *not* `\edef` to avoid bad surprises as far as this is possible, or use the new interfaces (`\TopMark`, `\FirstMark`, and `\LastMark`) which return the `(text)` in `\exp_not:n` to prevent uncontrolled expansion.

6.4.2 Legacy interface extensions

The new implementation adds three mark classes: `2e-left`, `2e-right` and `2e-right-nonempty` and patches `\markboth` and `\markright` slightly so that they also update these new mark classes, so that the new classes work with existing document classes.

As a result you can use `\LastMark{2e-left}` and `\FirstMark{2e-right}` instead of `\leftmark` and `\rightmark`. But more importantly, you can use any of the other retrieval commands to get a different status value from those marks, e.g., `\LastMark{2e-right}` would return the last subsection on the page (instead of the first as returned by `\rightmark`).

The difference between `2e-right` and `2e-right-nonempty` is that the latter will only be updated if the material for the mark is not empty. Thus `\markboth{title}{}` as issued by, say, `\sectionmark`, sets a `2e-left` mark with `title` and a `2e-right` mark with the empty string but does not add a `2e-right-nonempty` mark.

Thus, if you have a section at the start of a page and you would ask for `\FirstMark{2e-right}` you would get an empty string even if there are subsections on that page. But `2e-right-nonempty` would then give you the first or last subsection

on that page. Of course, nothing is simple. If there are no subsections it would tell you the last subsection from an earlier page. We therefore need comparison tools, e.g., if top and first are identical you know that the value is bogus, i.e., a suitable implementation would be

```
\IfMarksEqualTF{2e-right-nonempty}{top}{first}
  { <appropriate action if there was no real mark> }
  {\FirstMark{2e-right-nonempty}}
```

6.5 Notes on the mechanism

In contrast to vanilla T_EX, ε -T_EX extends the mark system to allow multiple independent marks. However, it does not solve the `\topmark` problem which means that L^AT_EX still needs to manage marks almost independently of T_EX. The reason for this is that the more complex output routine used by L^AT_EX to handle floats (and related structures) means that `\topmark(s)` remain unreliable. Each time the output routine is fired up, T_EX moves `\botmark` to `\topmark`, and while ε -T_EX extends this to multiple registers the fundamental concept remains the same. That means that the state of marks needs to be tracked by L^AT_EX itself. An early implementation of this package used T_EX's `\botmark` only to ensure the correct interaction with the output routine (this was before the ε -T_EX mechanism was even available). However, other than in a prototype implementation for L^AT_EX3, this package was never made public.

The new implementation now uses ε -T_EX's marks as they have some advantages, because with them we can leave the mark text within the galley and only extract the marks during the output routine when we are finally shipping out a page or storing away a column for use in the next page. That means we do not have to maintain a global data structure that we have to keep in sync with informational marks in the galley but can rely on everything being in one place and thus manipulations (e.g. reordering of material) will take the marks with them without a need for updating a fragile linkage.

To allow for completely independent marks we use the following procedure:

- For every type of marks we allocate a mark class so that in the output routine T_EX can calculate for each class the current top, first, and bottom mark independently. For this we use `\newmarks`, i.e., one marks register per class.
- As already mentioned firing up an output routine without shipping out a page means that T_EX's top marks get wrong so it is impossible to rely on T_EX's approach directly. What we do instead is to keep track of the real marks (for the last page or more generally last region) in some global variables.
- These variables are updated in the output routine at defined places, i.e., when we do real output processing but not if we use special output routines to do internal housekeeping.
- The trick we use to get correctly updated variables is the following: the material that contains new marks (for example the page to be shipped out) is stored in a box. We then use T_EX primitive box splitting functions by splitting off the largest amount possible (which should be the whole box if nothing goes really wrong). While that seems a rather pointless thing to do, it has one important side effect: T_EX sets up first and bottom marks for each mark class from the material it has split off. This way we get the first and last marks (if there have been any) from the material in the box.

- The top marks are simply the last marks from the previous page or region. And if there hasn't been a first or bottom mark in the box then the new top mark also becomes new first and last mark for that class.
- That mark data is then stored in global token lists for use during the output routine and legacy commands such as `\leftmark` or new commands such as `\TopMark` simply access the data stored in these token lists.

That's about it in a nutshell. Of course, there are some details to be taken care of—those are discussed in the implementation sections.

6.6 Public interfaces for packages such as `multicol`

The functions in this section are public so that packages can make use of them. However, this must be done with great care, e.g., `\mark_update_structure_from_material:nn` updates the global mark structure and can therefore be used only in places where such an update is meaningful, e.g., in special output routines. Elsewhere, a change to the mark structure would break the whole mechanism and querying the marks would return incorrect data.

`\mark_update_structure_from_material:nn` `\mark_update_structure_from_material:nn` $\langle region \rangle$ $\langle material\ with\ marks \rangle$

Helper function that inspects the marks inside the second argument and assigns new mark values based on that to the $\langle region \rangle$ given in the first argument. For this it first copies the mark structure from $\langle region \rangle$ to `previous- $\langle region \rangle$` and then takes all last mark values currently in the region and makes them the new top mark values. Finally it assigns new first and last values for all mark classes based on what was found in the second argument.

As a consequence, the allowed values for $\langle region \rangle$ are `page` and `column` because only they have `previous-...` counterparts.

Another important aspect to keep in mind is that marks are recognized only if they appear on the top level, e.g., if we want to process material stored in boxes we need to put it unboxed (using `\unvcopy` etc.) into the second argument.

`\mark_copy_structure:nn` `\mark_copy_structure:nn` $\langle alias \rangle$ $\langle source \rangle$

Helper function that copies all mark values in the $\langle source \rangle$ region to $\langle alias \rangle$, i.e., make the structures identical. Used to update the `previous-...` structures inside `\mark_update_structure_from_material:nn` and `first-column` and `last-column` structures inside the internal commands `__mark_update_singlecol_structures:` or `__mark__update_dbcol_structures:`.

`\mark_set_structure_to_err:n` `\mark_set_structure_to_err:n` $\langle region \rangle$

Helper function that sets all mark values in the $\langle region \rangle$ to an error message. This is currently used for `last-column` at times where using marks from it would be questionable/wrong, i.e., when we have just processed the first column in a two-column document.

`\mark_clear_structure:n` `\mark_clear_structure:n` $\langle region \rangle$

Helper function that sets all mark values in the $\langle region \rangle$ to empty. This is currently used for `column` when a multicol environment starts; this is because it wouldn't make sense if the top mark in the first column returned the last mark from a previous multicol (which may have been much earlier, with intermediate material).

`\mark_get_marks_for_reinsertion:nNN` `\mark_get_marks_for_reinsertion:nNN` $\langle source \rangle$
(token-list-var for collecting first marks)
(token-list-var for collecting last marks)

Helper function for extracting marks that would otherwise get lost, for example when they are hidden inside a box. This helper does not update mark structures and can therefore be used outside the output routine as well.

It collects all the top-level marks from inside the $\langle source \rangle$ and then adds suitable `\mark_insert:nn` commands to each of the two token lists. These token lists can then be executed at the right place to reinsert the marks, e.g., directly after the box. This is, for example, going to be used³⁷ by `multicol` when a short balanced `multicols` is returned to the galley for typesetting.

If the $\langle source \rangle$ consists of a single vertical box (plus possibly followed by some glue but nothing else) then the box is unpacked and the top-level marks are collected from its content. However, if it is not a vertical box or there are other data then nothing is unpacked and you have to do the unpacking yourself to get at the marks inside.

It is quite likely that one only needs a single token list for returning the `\mark_insert:nn` statements. If that is the case this command may change to take only two arguments.

6.7 Internal functions for the standard output routine of L^AT_EX

The functions in this section are tied to the output routine and used in the interface to L^AT_EX 2_ε and perhaps at some later time within a new output routine for L^AT_EX. They are not (yet) meant for general use and are therefore made internal, even though we already use them in `multicol`. Internal means that `@@` automatically gets replaced in the code (and in the documentation) so we have to give it a suitable value.

₁ $\langle @@=mark \rangle$

`_mark_update_singlecol_structures:` `_mark_update_singlecol_structures:`

L^AT_EX 2_ε integration function in case we are doing single column layouts. It assumes that the page content is already stored in `\@outputbox` and processes the marks inside that box. It is called as part of `\@opcol`.

`_mark_update dblcol_structures:` `_mark_update_singlecol_structures:`

L^AT_EX 2_ε integration function mark used when we are doing double column documents. It assumes that the page content is already stored in `\@outputbox` and processes the marks inside that box. It then does different post-processing depending on the start of the switch `\if@firstcolumn`. If we are in the second column it also has to update page marks, otherwise it only updates column marks. It too is called as part of `\@opcol`.

³⁷Probably not before 2025, though.

Chapter 7

Recording and cross-referencing document properties

7.1 Introduction

The module allows to record the “current state” of various document properties (typically the content of macros and values of counters) and to access them in other places through a label. The list of properties that can be recorded and retrieved are not fix and can be extended by the user. The values of the properties are recorded in the `.aux` file and can be retrieved at the second compilation.

The module uses the ideas of properties and labels. A label is a document reference point: a name for the user. An property is something that L^AT_EX can track, such as a page number, section number or name. The names of labels and properties may be arbitrary. Note that there is a single namespace for each.

7.2 Design discussion

The design here largely follows ideas from `zref`. In particular, there are two independent concepts: properties that can be recorded between runs, and labels which consist of lists of these properties. The reason for the split is that individual labels will want to record some but not all properties. For examples, a label concerned with position would track the x and y coordinates of the current point, but not for example the page number.

In the current implementation, properties share a single namespace. This allows multiple lists to re-use the same properties, for example page number, absolute page number, etc. This does mean that *changing* a standard property is an issue. However, some properties have complex definitions (again, see `zref` at present): having them in a single shared space avoids the need to copy code.

Labels could be implemented as `prop` data. That is not done at present as there is no obvious need to map to or copy the data. As such, faster performance is available using a hash table approach as in a “classical” set up. Data written to the `.aux` file uses simple paired *balanced text* not keyvals: this avoids any restrictions on names and again offers increased performance.

The `expl3` versions of the label command do not use `\@bsphack/\@esphack` to avoid double spaces, but the `LATEX 2ε` command does as it lives at the document command level.

The reference commands are expandable.

Currently the code has nearly no impact on the main `\label` and `\ref` commands as too many external packages rely on the concrete implementation. There is one exception: the label names share the same namespace. That means that if both `\label{ABC}` and `\RecordProperties{ABC}{page}` are used there is a warning Label ‘ABC’ multiply defined.

7.3 Handling unknown labels and properties

With the standard `\label/\ref` commands the requested label is either in the `.aux`-file (and so known) or not. In the first case the stored value can be used, in the second case the reference commands print two question marks.

With flexible property lists a reference commands asks for the value of a specific property stored under a label name and we have to consider more variants:

- If the requested property is unknown (not declared) the system is not correctly set up and an error is issued.
- If the label is unknown, the default of the property is used.
- If the label is known, but doesn’t provide a value for the property then again the default of the property is used.
- The command `\property_ref:nnn` allows to give a local default which is used instead of the property default in the two cases before.

7.4 Rerun messages

As the reference commands are expandable they can neither issue a message that the label or the label-property combination is unknown, nor can they trigger the rerun message at the end of the `LATEX` run.

Where needed such messages must therefore be triggered manually. For this two commands are provided: `\property_ref_undefined_warn:` and `\property_ref_undefined_warn:nn`. See below for a description.

7.5 Open points

- The `xpos` and `ypos` properties require that the position is stored first but there is no (public) engine independent interface yet. Code must use `\tex_savepos:D`.

7.6 Code interfaces

<code>\property_new:nnnn</code>	<code>\property_new:nnnn {⟨property⟩} {⟨setpoint⟩} {⟨default⟩} {⟨code⟩}</code>
<code>\property_gset:nnnn</code>	<code>\property_gset:nnnn {⟨property⟩} {⟨setpoint⟩} {⟨default⟩} {⟨code⟩}</code>

L^AT_EX 2_ε-interface: see `\NewProperty`, `\SetProperty`.

Sets the `⟨property⟩` to have the `⟨default⟩` specified, and at the `⟨setpoint⟩` (either `now` or `shipout`) to write the result of the `⟨code⟩` as part of a label. The `⟨code⟩` should be expandable. The expansion of `⟨code⟩` (the value of the property) is written to the `.aux` file and read back from there at the next compilation. Values should assume that the standard L^AT_EX catcode régime with `@` a letter is active then.

If the property is declared within a package it is suggested that its name is build from letters, hyphens and slashes, and is always structured as follows:
`⟨package-name⟩/⟨property-name⟩`.

<code>\property_record:nN</code>	<code>\property_record:nN {⟨label⟩} {⟨clist var⟩}</code>
<code>\property_record:nn</code>	<code>\property_record:nn {⟨label⟩} {⟨clist⟩}</code>

`\property_record:(nV|ee)`

L^AT_EX 2_ε-interface: see `\RecordProperties`.

Writes the list of properties given by the `⟨clist⟩` to the `.aux` file with the `⟨label⟩` specified.

<code>\property_ref:nn *</code>	<code>\property_ref:nn {⟨label⟩} {⟨property⟩}</code>
---------------------------------	--

`\property_ref:ee *`

L^AT_EX 2_ε-interface: see `\RefProperty`.

Expands to the value of the `⟨property⟩` for the `⟨label⟩`, if available, and the default value of the property otherwise. If `⟨property⟩` has not been declared with `\property_new:nnnn` an error is issued. The command raises an internal, expandable, local flag if the reference can not be resolved.

<code>\property_item:nn *</code>	<code>\property_item:nn {⟨label⟩} {⟨property⟩}</code>
----------------------------------	---

`\property_item:ee *`

New: 2025-11-20

Retrieves the value of the `⟨property⟩` for the `⟨label⟩` like `\property_ref:nn` but the result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `⟨value⟩` does not expand further when appearing in an `e`-type or `x`-type argument expansion. This allows for example to handle values containing user commands which are not safe in an expansion context.

<code>\property_ref:nnn *</code>	<code>\property_ref:nnn {⟨label⟩} {⟨property⟩} {⟨local default⟩}</code>
----------------------------------	---

`\property_ref:een *`

L^AT_EX 2_ε-interface: see `\RefProperty`.

Expands to the value of the `⟨property⟩` for the `⟨label⟩`, if available, and to `⟨local default⟩` otherwise. If `⟨property⟩` has not been declared with `\property_new:nnnn` an error is issued. The command raises an internal, expandable local flag if the reference can not be resolved.

<code>\property_ref_undefined_warn:</code>	<code>\property_ref_undefined_warn:</code>
--	--

L^AT_EX 2_ε-interface: not provided.

Triggers the standard warning

LaTeX Warning: There were undefined references.

at the end of the document if there was a recent `\property_ref:nn` or `\property_ref:nnn` which couldn't be resolved and so raised the flag. "Recent" means in the same group or in some outer group!

```
\property_ref_undefined_warn:n \property_ref_undefined_warn:n {<label>}
\property_ref_undefined_warn:e
```

L^AT_EX 2_ε-interface: not provided.
Triggers the standard warning
LaTeX Warning: There were undefined references.
at the end of the document if `<label>` is not known. At the point where it is called it also issues the warning
Reference ‘`<label>`’ on page `<page>` undefined.

```
\property_ref_undefined_warn:nn \property_ref_undefined_warn:nn {<label>} {<property>}
\property_ref_undefined_warn:ee
```

L^AT_EX 2_ε-interface: see `\RefUndefinedWarn`.
Triggers the standard warning
LaTeX Warning: There were undefined references.
at the end of the document if the reference can not be resolved. At the point where it is called it also issues the warning
Reference ‘`<label>`’ on page `<page>` undefined
if the label is unknown, or the more specific
Property ‘`<property>`’ undefined for reference ‘`<label>`’ on page `<page>`
if the label is known but doesn’t provide a value for the requested property.

```
\property_if_exist_p:n * \property_if_exist_p:n {<property>}
\property_if_exist_p:e * \property_if_exist:nTF {<property>} {<true code>} {<false code>}
\property_if_exist:nTF * LATEX 2ε-interface: \IfPropertyExistsTF.
\property_if_exist:eTF * Tests if the <property> has been declared.
```

```
\property_if_recorded_p:n * \property_if_recorded_p:n {<label>}
\property_if_recorded_p:e * \property_if_recorded:nTF {<label>} {<true code>} {<false code>}
\property_if_recorded:nTF *
\property_if_recorded:eTF *
```

L^AT_EX 2_ε-interface: `\IfLabelExistsTF`
Tests if the `<label>` is known. This is also true if the label has been set with the standard `\label` command.

```
\property_if_recorded_p:nn * \property_if_recorded_p:nn {<label>} {<property>}
\property_if_recorded_p:ee * \property_if_recorded:nnTF {<label>} {<property>} {<true code>} {<false code>}
\property_if_recorded:nnTF *
\property_if_recorded:eeTF *
```

L^AT_EX 2_ε-interface: `\IfPropertyRecordedTF`.
Tests if the label `<label>` is known and if it provides a value of the `<property>`.

7.7 Auxiliary file interfaces

```
\new@label@record \new@label@record {<label>} {<data>}
```

This is a command only for use in the `.aux` file. It loads the key–value list of `<data>` to be available for the `<label>`.

7.8 L^AT_EX 2_ε interface

The LaTeX interfaces always expand label and property arguments. This means that one must be careful when using active chars or commands in the names. UTF8-chars are protected and should be safe, similar most babel shorthands.

<code>\NewProperty</code>	<code>\NewProperty {⟨property⟩} {⟨setpoint⟩} {⟨default⟩} {⟨code⟩}</code>
<code>\SetProperty</code>	<code>\SetProperty {⟨property⟩} {⟨setpoint⟩} {⟨default⟩} {⟨code⟩}</code>

Sets the `⟨property⟩` to have the `⟨default⟩` specified, and at the `⟨setpoint⟩` (either `now` or `shipout`) to write the result of the `⟨code⟩` as part of a label. The `⟨code⟩` should be expandable. The expansion of `⟨code⟩` (the value of the property) is written to the `.aux` file and read back from there at the next compilation (at which point normally the standard L^AT_EX catcode régime with `@` a letter is active).

<code>\RecordProperties</code>	<code>\RecordProperties {⟨label⟩} {⟨clist⟩}</code>
--------------------------------	--

Writes the list of properties given by the `⟨clist⟩` to the `.aux` file with the `⟨label⟩` specified. Similar to the standard `\label` command the arguments are expanded. So `⟨clist⟩` can be a macro containing a list of properties. Also similar to the standard `\label` command, the command is surrounded by an `\@bsphack/\@esphack` pair to preserve spacing.

<code>\RefProperty *</code>	<code>\RefProperty [⟨local default⟩] {⟨label⟩} {⟨property⟩}</code>
-----------------------------	--

Expands to the value of the `⟨property⟩` for the `⟨label⟩`, if available, and the default value of the property or – if given – to `⟨local default⟩` otherwise. If `{⟨property⟩}` has not been declared an error is issued.

<code>\IfPropertyExistsTF</code>	<code>\IfPropertyExistsTF {⟨property⟩} {⟨true code⟩} {⟨false code⟩}</code>
<code>\IfPropertyExistsT</code>	Tests if the <code>⟨property⟩</code> has been declared.
<code>\IfPropertyExistsF</code>	

<code>\IfLabelExistsTF</code>	<code>\IfLabelExistsTF {⟨label⟩} {⟨true code⟩} {⟨false code⟩}</code>
-------------------------------	--

<code>\IfLabelExistsT</code>	Tests if the <code>⟨label⟩</code> has been recorded. This is also true if a label has been set with the standard <code>\label</code> command.
<code>\IfLabelExistsF</code>	

<code>\IfPropertyRecordedTF</code>	<code>\IfPropertyRecordedTF {⟨label⟩} {⟨property⟩} {⟨true code⟩} {⟨false code⟩}</code>
------------------------------------	--

<code>\IfPropertyRecordedT</code>	Tests if the label and a value of the <code>⟨property⟩</code> for the <code>⟨label⟩</code> are both known.
<code>\IfPropertyRecordedF</code>	

<code>\RefUndefinedWarn</code>	<code>\RefUndefinedWarn {⟨label⟩} {⟨property⟩}</code>
--------------------------------	---

Triggers the standard warning

LaTeX Warning: There were undefined references.
at the end of the document if the reference for `⟨label⟩` and `⟨property⟩` can not be resolved. At the point where it is called it also issues the warning

Reference ‘`⟨label⟩`’ on page `⟨page⟩` undefined
if the label is unknown, or the more specific

Property ‘`⟨property⟩`’ undefined for reference ‘`⟨label⟩`’ on page `⟨page⟩` if the label is known but doesn’t provide a value for the requested property.

7.9 Pre-declared properties

abspage (shipout) The absolute value of the current page: starts at 1 and increases monotonically at each shipout.

page (shipout) The current page as given by `\thepage`: this may or may not be a numerical value, depending on the current style. Contrast with `\abspage`. You get this value also with the standard `\label/\pageref`.

pagenum (shipout) The current page as arabic number. This is suitable for integer operations and comparisons.

label (now) The content of `\@currentlabel`. This is the value that you get also with the standard `\label/\ref`.

title (now) The content of `\@currentlabelname`. This command is filled beside others by the `nameref` package and some classes (e.g. `memoir`).

target (now) The content of `\@currentHref`. This command is normally filled by for example `hyperref` and gives the name of the last destination it created.

pagetarget (shipout) The content of `\@currentHpage`. This command is filled for example by a recent version of `hyperref` and then gives the name of the last page destination it created.

counter (now) The content of `\@currentcounter`. This command contains after a `\refstepcounter` the name of the counter.

xpos (shipout) This stores the x and y coordinates of a point previously stored with ypos `\pdfsavepos/\savepos`. E.g. (if `bidi` is used it can be necessary to save the position before and after the label):

```
\tex_savepos:D
\property_record:nn{myposition}{xpos,ypos}
\tex_savepos:D
```

Part III
Design-level tools

Chapter 8

L^AT_EX's socket management

8.1 Introduction

A L^AT_EX source file is transformed into a typeset document by executing code for each command or environment in the document source. Through various steps this code transforms the input and eventually generates typeset output appearing in a “galley” from which individual pages are cut off in an asynchronous way. This page generating process is normally not directly associated with commands in the input³⁸ but is triggered whenever the galley has received enough material to form another page (giving current settings).

As part of this transformation input data may get stored in some form and later reused, for example, as part of the output routine processing.

8.2 Configuration of the transformation process

There are three different major methods offered by L^AT_EX to configure the transformation process:

- through the template mechanism,
- through the hook mechanism, or
- through sockets and plugs.

They offer different possibilities (with different features and limitations) and are intended for specific use cases, though it is possible to combine them.

8.2.1 The template mechanism

The template mechanism is intended for more complex document-level elements (e.g., headings such as `\section` or environments like `itemize`). The template code implements the overall processing logic for such an element and offers a set of parameters to influence the final result.

The document element is then implemented by (a) selecting a suitable template (there may be more than one available for the kind of document element) and (b) by

³⁸Excepts for directives such as `\newpage`.

setting its parameters to desired values. This then forms a so-called instance which is executed when the document element is found in the source.

By altering the parameter values (in a document class or in the document preamble) or, if more drastic layout changes are desired, by selecting a different template and then adjusting its parameters, a wide variety of layouts can be realized through simple configuration setups without the need to develop new code.

The target audience of this method are therefore document class developers or users who wish to alter an existing layout (implemented by a document class) in certain (minor) ways.

The template mechanism is currently documented as part of the `xtemplate` package and one more elaborate implementation can be found as part of the `latex-lab` code for lists (to be documented further).

8.2.2 The hook mechanism

Hooks are places in the kernel code (or in packages) that offer packages the possibility to inject additional code at specific points in the processing in a controlled way without the need to replace the existing code block (and thereby overwriting modifications/extensions made by other packages). The target audience is therefore mainly package developers, even though some hooks can be useful for document authors.

Obviously, what can reasonably be added into a hook depends on the individual hook (hopefully documented as part of the hook documentation), but in general the idea behind hooks is that more than one package could add code into the hook at the same time. Perhaps the most famous hook (that L^AT_EX had for a very long time) is `begindocument` into which many packages add code to through `\AtBeginDocument{<code>}` (which is nowadays implemented as a shorthand for `\AddToHook{begindocument}{<code>}`). To resolve possible conflicts between injections by different packages there is a rule mechanism by which code chunks in a hook can be ordered in a certain way and by which incompatible packages can be detected if a resolution is impossible.

In contrast to template code, there is no standard configuration method through parameters for hooks, i.e., the code added to a hook “is” the configuration. If it wants to provide for configuration through parameters it has to also provide its own method to set such parameters in some way. However, in that case it is likely that using a hook is not the right approach and the developer better calls a template instance instead which then offers configuration through a key/value interface.

In most cases, hooks do not take any arguments as input. Instead, the data that they can (and are allowed to) access depends on the surrounding context.

For example, the various hooks available during the page shipout process in L^AT_EX’s output routine can (and have to) access the accumulated page material stored in a box named `\ShipoutBox`. This way, code added to, say, the `shipout/before` hook could access the page content, alter it, and then write it back into `\ShipoutBox` and any other code added to this hook could then operate on the modified content. Of course, for such a scheme to work the code prior to executing the hook would need to setup up data in appropriate places and the hook documentation would need to document what kind of storage can be accessed (and possibly altered) by the hook.

There are also hooks that take arguments (typically portions of document data) and in that case the hook code can access these arguments through `#1`, `#2`, etc.

The hook mechanism is documented in `lthooks-doc.pdf`.

8.2.3 The socket mechanism

In some cases there is code that implements a certain programming logic (for example, combining footnotes, floats, and the text for the current page to be shipped out) and if this logic should change (e.g., footnotes to be placed above bottom floats instead of below) then this whole code block needs to be replaced with different code.

In theory, this could be implemented with templates, i.e., the code simply calls some instance that implements the logic and that instance is altered by selecting a different templates and/or adjusting their parameters. However, in many cases customization through parameters is overkill in such a case (or otherwise awkward, because parameterization is better done on a higher level instead of individually for small blocks of code) and using the template mechanism just to replace one block of code with a different one results in a fairly high performance hit. It is therefore usually not a good choice.

In theory, it would also be possible to use a hook, but again that is basically a misuse of the concept, because in this use case there should never be more than one block of code inside the hook; thus, to alter the processing logic one would need to set up rules that replace code rather than (as intended) execute all code added to the hook.

For this reason L^AT_EX now offers a third mechanism: “sockets” into which one can place exactly one code block — a “plug”.

In a nutshell: instead of having a fixed code block somewhere as part of the code, implementing a certain programming logic there is a reference to a named socket at this point. This is done by first declaring the named socket with:

```
\NewSocket{<socket-name>}{<number-of-inputs>}
```

This is then referenced at the point where the replaceable code block should be executed with:

```
\UseSocket{<socket-name>}
```

or, if the socket should take a number of inputs (additional arguments beside the name) with

```
\UseSocket{<socket-name>}{<arg1>}...{<argnumber-of-inputs>}
```

In addition, several code blocks (a.k.a. plugs) implementing different logic for this socket are set up, each with a declaration of the form:

```
\NewSocketPlug{<socket-name>}{<socket-plug-name>}{<code>}
```

Finally, one of them is assigned to the socket:

```
\AssignSocketPlug{<socket-name>}{<socket-plug-name>}
```

If the programming logic should change, then all that is necessary is to make a new assignment with `\AssignSocketPlug` to a different `{<socket-plug-name>}`. This assignment obeys scope so that an environment can alter a socket without the need to restore the previous setting manually.

If the socket takes inputs, then those need to be provided to `\UseSocket` and in that case they can be referenced in the `<code>` argument of `\NewSocketPlug` with `#1`, `#2`, etc.

In most cases a named socket is used only in a single place, but there is, of course, nothing wrong with using it in several places, as long as the code in all places is supposed to change in the same way.

Examples

We start by declaring a new socket named `foo` that expects two inputs:

```
\NewSocket{foo}{2}
```

Such a declaration has to be unique across the whole $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ run. Thus, if another package attempts to use the same name (regardless of the number of inputs) it will generate an error:

```
\NewSocket{foo}{2}
\NewSocket{foo}{1}
```

Both declarations would therefore produce:

```
! LaTeX socket Error: Socket 'foo' already declared!
```

You also get an error if you attempt to declare some socket plug and the socket name is not yet declared, e.g.,

```
\NewSocketPlug{baz}{undeclared}{some code}
```

generates

```
! LaTeX socket Error: Socket 'baz' undeclared!
```

Setting up plugs for the socket is done like this:

```
\NewSocketPlug{foo}{plug-A}
  {\begin{quote}\itshape foo-A: #1!#2\end{quote}}
\NewSocketPlug{foo}{plug-B}
  {\begin{quote}\sffamily foo-B: #2\textsuperscript{2}\end{quote}}
```

This will set up the plugs `plug-A` and `plug-B` for this socket.

We still have to assign one or the other to the socket, thus without doing that the line

```
\UseSocket{foo}{hello}{world}
```

produces nothing because the default plug for sockets with 2 inputs is `noop` (which grabs the additional arguments and throws them away).³⁹

So let's do the assignment

```
\AssignSocketPlug{foo}{plug-A}
```

and then

```
\UseSocket{foo}{hello}{world}
```

will properly typeset

```
foo-A: hello!world
```

and after

```
\AssignSocketPlug{foo}{plug-B}
```

³⁹If socket `foo` would have been a socket with one input, then the default plug would be `identity`, in which case the socket input would remain without braces and gets typeset!

and another call to

```
\UseSocket{foo}{hello}{world}
```

we get

```
foo-B: world2
```

If we attempt to assign a plug that was not defined, e.g.,

```
\AssignSocketPlug{foo}{plug-C}
```

then we get an error during the assignment

```
! LaTeX socket Error: Plug 'plug-C' for socket 'foo' undeclared!
```

and the previous assignment remains in place.

To see what is known about a socket and its plugs you can use `\ShowSocket` or `\LogSocket` which displays information similar to this on the terminal or in the transcript file:

```
Socket foo:
  number of inputs = 2
  available plugs = noop, plug-A, plug-B
  current plug = plug-B
  definition = \long macro:#1#2->\begin {quote}\sffamily
foo-B: #2\textsuperscript {2}\end {quote}
```

Details and semantics

In this section we collect some normative statements.

- From a functional point of view sockets are like simple $\text{T}_{\text{E}}\text{X}$ macros, i.e., they expect 0 to 9 mandatory arguments (the socket inputs) and get replaced by their “expansion”
- A socket is “named” and the name consists of ASCII letters [a-z], [A-Z], [0-9], [-/@] only
- Socket names have to be unique, i.e., there can be only one socket named $\langle name \rangle$. This is ensured by declaring each socket with `\NewSocket`.

However, there is no requirement that sockets and hook names have to be different. In fact, if a certain action that could otherwise be specified as hook code has to be executed always last (or first) one could ensure this by placing a socket (single action) after a hook (or vice versa) and using the same name to indicate the relationship, e.g.,

```
\UseHook{foo}           % different package can add code here
\UseSocket{foo}        % only one package can assign a plug
```

This avoids the need to order the hook code to ensure that something is always last.

- Best practice naming conventions are ... *to be documented*

- A socket has documented inputs which are
 - the positional arguments (if any) with a description of what they contain when used
 - implicit data (registers and other 2e/expl3 data stores) that the socket is allowed to make use of, with a documented description of what they contain (if relevant for the task at hand—no need to describe the whole L^AT_EX universe)
 - information about the state of the T_EX engine (again when relevant), e.g. is called in mmode or vmode or in the output routine or ...
 - ... anything missing?
- A socket has documented results/outputs which can be
 - what kind of data it should write to the current list (if that is part of its task)
 - what kind of registers and other 2e/expl3 data stores it should modify and in what way
 - what kind of state changes it should do (if any)
 - ... *anything else?*
- At any time a socket has one block of code (a plug :-)) associated with it. Such code is itself named and the association is done by linking the socket name to the code name (putting a plug into the socket).
- The name of a plug consists of ASCII letters [a-z], [A-Z], [0-9], [-/@] only.
- Socket plug names have to be unique within on a per socket basis, but it is perfectly allowed (and sensible in some cases) to use the same plug name with different sockets (where based on the sockets' purposes, different actions may be associated with the plug name). For example `noop` is a plug name declared for every socket, yet its action “grab the socket inputs and throw them away” obviously differs depending on how many inputs the socket has.
- When declaring a plug it is stated for which socket it is meant (i.e., its code can only be used with that socket). This means that the same plug name can be used with different sockets referring to different code in each case.
- Configuration of a socket can only be done by linking different code to it. Nevertheless the code linked to it can provide its own means of configuration (but this is outside of the spec).
- Technically execution of a socket (`\UseSocket`) involves
 - doing any house keeping (like writing debugging info, ...);
 - looking up the current code association (what plug is in the socket);
 - executing this code which will pick up the mandatory arguments (happens at this point, not before), i.e., it is like calling a csname defined with


```
\def\foo#1#2...{...#1...#2...}
```
 - do some further house keeping (if needed).
- A socket is typically only used in one place in code, but this is not a requirement, i.e., if the same operation with the same inputs need to be carried out in several places the same named socket can be used.

Command syntax

We give both the L^AT_EX 2_ε and the L3 programming layer command names.

<code>\NewSocket</code>	<code>\NewSocket</code>	<code>{\socket-name}</code>	<code>{\number-of-inputs}</code>
<code>\socket_new:nn</code>	<code>\socket_new:nn</code>	<code>{\socket-name}</code>	<code>{\number-of-inputs}</code>

Declares a new socket with name `\socket-name` having `\number-of-inputs` inputs. There is automatically a plug `noop` declared for it, which does nothing, i.e., it gobbles the socket inputs (if any). This is made the default plug except for sockets with one input which additionally define the plug `identity` and assign that as their default.

This `identity` plug simply returns the socket input without its outer braces. The use case for this plug are situations like this:

```
\UseSocket{taggsupport/footnote}{\code}
```

If tagging is not active and the socket contains the plug `identity` then this returns `\code` without the outer braces and to activate tagging all that is necessary is to change the plug to say `tagpdf` so that it surrounds `\code` by some tagging magic. This is the most common use case for sockets with one input, which is why they have this special default.

The socket documentation should describe its purpose, its inputs and the expected results as discussed above.

The declaration is only allowed at top-level, i.e., not inside a group.

<code>\NewSocketPlug</code>	<code>\NewSocketPlug</code>	<code>{\socket-name}</code>	<code>{\socket-plug-name}</code>	<code>{\code}</code>
<code>\socket_new_plug:nnn</code>	<code>\socket_new_plug:nnn</code>	<code>{\socket-name}</code>	<code>{\socket-plug-name}</code>	<code>{\code}</code>
<code>\socket_set_plug:nnn</code>	<code>\socket_set_plug:nnn</code>	<code>{\socket-name}</code>	<code>{\socket-plug-name}</code>	<code>{\code}</code>

Declares a new plug for socket `\socket-name` that runs `\code` when executing. It complains if the plug was already declared previously.

The form `\socket_set_plug:nnn` changes an existing plug. As this should normally not be necessary, we currently have only an L3 layer name for the few cases it might be useful.

The declarations can be made inside a group and obey scope, i.e., they vanish if the group ends.

<code>\AssignSocketPlug</code>	<code>\AssignSocketPlug</code>	<code>{\socket-name}</code>	<code>{\socket-plug-name}</code>
<code>\socket_assign_plug:nn</code>	<code>\socket_assign_plug:nn</code>	<code>{\socket-name}</code>	<code>{\socket-plug-name}</code>

Assigns the plug `\socket-plug-name` to the socket `\socket-name`. It errors if either socket or plug is not defined.

The assignment is local, i.e., it obeys scope.

<code>\UseSocket</code>	<code>\UseSocket</code>	<code>{\socket-name}</code>
<code>\socket_use:nw</code>	<code>\socket_use:nnn</code>	<code>{\socket-name}</code> <code>{\socket-arg₁}</code> <code>{\socket-arg₂}</code>
<code>\socket_use:n</code>	Executes the socket <code>\socket-name</code> by retrieving the <code>\code</code> of the current plug assigned to the socket. This is the only command that would appear inside macro code in packages.	
<code>\socket_use:nn</code>	For performance reasons there is no explicit check that the socket was declared!	
<code>\socket_use:nnn</code>	The different L3 programming layer commands are really doing the same thing: they grab as many arguments as defined as inputs for the socket and then pass them to the plug. The different names are only there to make the code more readable, i.e., to indicate how many arguments are grabbed in total (note that no runtime check is made to verify that this is actually true). We only provide them for sockets with up to 3 inputs (most likely those with zero or one input would have been sufficient). If you happen to have a socket with more inputs, use <code>\socket_use:nw</code> .	
<code>\socket_use:nnnn</code>		

The different L3 programming layer commands are really doing the same thing: they grab as many arguments as defined as inputs for the socket and then pass them to the plug. The different names are only there to make the code more readable, i.e., to indicate how many arguments are grabbed in total (note that no runtime check is made to verify that this is actually true). We only provide them for sockets with up to 3 inputs (most likely those with zero or one input would have been sufficient). If you happen to have a socket with more inputs, use `\socket_use:nw`.

<code>\socket_use_expandable:nw</code>	*	<code>\socket_use_expandable:n</code>	<code>{\socket-name}</code>
<code>\socket_use_expandable:n</code>	*		

Fully expandable variant of `\socket_use:n`. This can be used in macro code to retrieve code from sockets which need to appear in an expandable context.

This usually requires the plug to only contain expandable code and should therefore only be used for sockets which are clearly documented to be used in an expandable context. This command does not print any debugging info when `\DebugSocketsOn` is active and should therefore be avoided whenever possible.

For performance reasons there is no explicit check that the socket was declared!

<code>\ShowSocket</code>	<code>\ShowSocket</code>	<code>{\socket-name}</code>
<code>\LogSocket</code>	<code>\socket_show:n</code>	<code>{\socket-name}</code>
<code>\socket_show:n</code>	Displays information about the socket <code>\socket-name</code> and its state then stops and waits for further instructions — at the moment some what rudimentary.	
<code>\socket_log:n</code>	<code>\LogSocket</code> and <code>\socket_log:n</code> only differ in that they don't stop.	

It is sometimes necessary/helpful to know if a particular socket or plug exists (or is assigned to a certain socket) and based on that take different actions.

<code>\IfSocketExistsTF</code>	*	<code>\IfSocketExistsTF</code>	<code>{\socket-name}</code> <code>{\true code}</code> <code>{\false code}</code>
<code>\socket_if_exist:nTF</code>	*		

If socket `\socket-name` exists then execute `\true code` otherwise `\false code`. Variants with only T or F are also available.

<code>\IfSocketPlugExistsTF</code>	*	<code>\IfSocketPlugExistsTF</code>	<code>{\socket-name}</code> <code>{\plug-name}</code>
<code>\socket_if_plug_exist:nnTF</code>	*		<code>{\true code}</code> <code>{\false code}</code>

If plug `\plug-name` for socket `\socket-name` exists then execute `\true code` otherwise `\false code`. Variants with only T or F are also available.

<code>\IfSocketPlugAssignedTF</code>	*	<code>\IfSocketPlugAssignedTF</code>	<code>{\socket-name}</code> <code>{\plug-name}</code>
<code>\socket_if_plug_assigned:nnTF</code>	*		<code>{\true code}</code> <code>{\false code}</code>

If plug `\plug-name` is assigned to socket `\socket-name` then execute `\true code` otherwise `\false code`. Variants with only T or F are also available.

<code>\DebugSocketsOn</code>	<code>\DebugSocketsOn ... \DebugSocketsOff</code>
<code>\DebugSocketsOff</code>	Turns debugging of sockets on or off.
<code>\socket_debug_on:</code>	
<code>\socket_debug_off:</code>	

Rationale for error handling

The errors during the declarations are produced to help with typos—after all, such declarations might be part of a document preamble (not that likely, but possible). However, `\UseSocket` is not doing much checking, e.g.,

```
\UseSocket{mispelled-socket}{hello}{world}
```

will generate a rather low-level error and then typesets “helloworld” because there is no dedicated runtime check if `mispelled-socket` is a known socket.

The reason is that if the misspelling is in the code, then this is a programming error in the package and for speed reasons \LaTeX does not repeatedly make runtime checks for coding errors unless they can or are likely to be user introduced.

Chapter 9

Templates: Prototype document functions

9.1 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 $\text{T}_{\text{E}}\text{X}$ 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 $\text{T}_{\text{E}}\text{X}$ implementation in the middle is the glue between the two.

$\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$'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 $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X} 2_{\epsilon}$ classes look somewhat dated now in terms of their visual design, their typography is generally sound (barring the occasional minor faults).

However, $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X} 2_{\epsilon}$ 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 `ltemplates` 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. `ltemplates` also makes it easier for $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ programmers to provide their own customizations on top of a pre-existing class.

9.2 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 9.4, 9.5, and 9.7, respectively.

9.3 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.

9.4 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.

9.5 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 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 <code>⟨choices⟩</code>
<code>commalist</code>	A comma-separated list
<code>function{⟨N⟩}</code>	A (protected) function definition with N arguments (N from 0 to 9)
<code>instance{⟨name⟩}</code>	An instance of type <code>⟨name⟩</code>
<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 9.6)
<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 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 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 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.

<code>\SetKnownTemplateKeys</code>	<code>\SetKnownTemplateKeys {<type>} {<template>} {<keyvals>}</code>
<code>\SetTemplateKeys</code>	<code>\SetTemplateKeys {<type>} {<template>} {<keyvals>}</code>
<code>\UnusedTemplateKeys</code>	<code>\UnusedTemplateKeys % all <keyvals> unused by previous \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` 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.⁴⁰

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

⁴⁰Using `\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.

9.6 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.

9.7 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`
 `{<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.

`\InstanceValue` ★ `\InstanceValue {<type>} {<instance>} {<key>}`

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

<code>\IfInstanceExistsT</code>	<code>\IfInstanceExistsTF {<type>} {<instance>} {<>true code>} {<>false code>}</code>
<code>\IfInstanceExistsF</code>	Tests if the named <code><instance></code> of a <code><type></code> exists, and then inserts the appropriate code into the input stream.
<code>\IfInstanceExistsTF</code>	

<code>\DeclareInstanceCopy</code>	<code>\DeclareInstanceCopy</code>
	<code>{<type>} {<instance2>} {<instance1>}</code>
	Copies the <code><values></code> for <code><instance1></code> for an <code><type></code> to <code><instance2></code> .

9.8 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.

<code>\UseInstance</code>	<code>\UseInstance</code>
	<code>{<type>} {<instance>} <arguments></code>
	Uses an <code><instance></code> of the <code><type></code> , which will require <code><arguments></code> as determined by the number specified for the <code><type></code> . The <code><instance></code> 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 <code><template></code> of the specified <code><type></code> , applying the <code><settings></code> and absorbing <code><arguments></code> as detailed by the <code><type></code> declaration. This in effect is the same as creating an instance using <code>\DeclareInstance</code> and immediately using it with <code>\UseInstance</code> , 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> }
}
```

9.9 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.

9.9.1 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 \LaTeX 2_{ϵ} 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
```

9.10 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.

Symbols	
/after (hook)	<i>32, 32, 34</i>
/before (hook)	<i>32, 32</i>
\<addto-cmd>	<i>5</i>
A	
A/foo (hook)	<i>13, 13</i>
\abspage	<i>77</i>
abspage	<i>77</i>
\ActivateGenericHook	<i>4</i>
\AddEverypageHook	<i>60</i>
\AddThispageHook	<i>60</i>
\AddToHook	<i>22</i>
\AddToHookNext	<i>57</i>
\AddToHookNextWithArguments	<i>8</i>
\AddToHookWithArguments	<i>22</i>
after (hook)	<i>27</i>
\AfterEndEnvironment	<i>27</i>
\Alph	<i>58</i>
\arabic	<i>58</i>
\AssignSocketPlug	<i>81</i>
\AssignTemplateKeys	<i>92</i>
\AtBeginDocument	<i>21</i>
\AtBeginDvi	<i>60</i>
\AtBeginEnvironment	<i>27</i>
\AtBeginShipout	<i>60</i>
\AtBeginShipoutAddToBox	<i>59</i>
\AtBeginShipoutAddToBoxForeground	<i>59</i>
\AtBeginShipoutBox	<i>59</i>
\AtBeginShipoutDiscard	<i>60</i>
\AtBeginShipoutFirst	<i>60</i>
\AtBeginShipoutInit	<i>60</i>
\AtBeginShipoutNext	<i>60</i>
\AtBeginShipoutOriginalShipout	<i>59</i>
\AtBeginShipoutUpperLeft	<i>59</i>
\AtBeginShipoutUpperLeftForeground	<i>59</i>
\AtEndDocument	<i>25</i>
\AtEndDvi	<i>56</i>
\AtEndEnvironment	<i>27</i>
\AtEndOfClass	<i>34</i>
\AtEndOfPackage	<i>34</i>
\AtEndPreamble	<i>28</i>
\AtNextShipout	<i>60</i>
B	
babel/<language>/afterextras (hook)	<i>22</i>
\BeforeBeginEnvironment	<i>27</i>
\begin	<i>28</i>
begindocument (hook)	<i>80, 80, 25, 25, 25, 28, 38</i>
begindocument/before (hook)	<i>28</i>
begindocument/end (hook)	<i>28</i>
\bfdefault	<i>30</i>
bfseries (hook)	<i>30, 30</i>
\bfseries	<i>30</i>
bfseries/defaults (hook)	<i>30, 30</i>
bool commands:	
\l_tmpa_bool	<i>92</i>
\botmark	<i>62</i>
\box	<i>47</i>
C	
\caption	<i>39</i>
\chaptermark	<i>68</i>
class (hook)	<i>22</i>
class/ (hook)	<i>34</i>
class/.../after	<i>34</i>
class/.../before	<i>34</i>
class/<name>/after (hook)	<i>34</i>
class/<name>/before (hook)	<i>34</i>
class/after (hook)	<i>34</i>
class/after	<i>34</i>
class/before (hook)	<i>34</i>
class/before	<i>34</i>
\ClearHookNext	<i>8</i>
\ClearHookRule	<i>12</i>
\clearpage	<i>34</i>
clist commands:	
\l_tmpa_clist	<i>92</i>
cmd (hook)	<i>16, 22, 22, 41, 42</i>
cmd/<name>/after (hook)	<i>27, 38, 41</i>
cmd/<name>/before (hook)	<i>27, 38, 38</i>
cmd/foo/before (hook)	<i>40</i>
cmd/include/after (hook)	<i>35</i>
cmd/include/before (hook)	<i>35</i>
cmd/section/after (hook)	<i>41, 41</i>
cmd/section/before (hook)	<i>41</i>
counter	<i>77</i>
\csname	<i>8</i>
\CurrentFile	<i>33</i>
\CurrentFilePath	<i>33</i>
\CurrentFilePathUsed	<i>33</i>

<code>\CurrentFileUsed</code>	33	exp commands:	
		<code>\exp_not:n</code>	68
		<code>expand@font@defaults</code> (hook)	30
		<code>\ExplSyntaxOn</code>	39
		F	
		<code>file</code> (hook)	13, 22
		file commands:	
		<code>\l_file_search_path_seq</code>	33
		<code>file/.../after</code>	32
		<code>file/.../before</code>	32
		<code>file/<file name>/after</code> (hook) ...	33, 33
		<code>file/<file name>/before</code> (hook) ..	33, 33
		<code>file/<filename>/after</code> (hook)	32
		<code>file/<filename>/before</code> (hook)	32
		<code>file/<package name>.sty/after</code> (hook)	34
		<code>file/<package name>.sty/before</code> (hook)	34
		<code>file/after</code> (hook)	32, 33, 34
		<code>file/after</code>	32
		<code>file/array.sty/after</code> (hook)	33
		<code>file/before</code> (hook)	32, 33, 34
		<code>file/before</code>	32
		<code>\FirstMark</code>	64
		<code>\firstmark</code>	62
		<code>foo</code> (socket)	82, 82
		fp commands:	
		<code>\l_tmpa_fp</code>	92
		G	
		<code>\glossary</code>	63
		H	
		<code>\hbox</code>	59
		hook commands:	
		<code>\hook_activate_generic:n</code>	16
		<code>\hook_debug_off:</code>	18
		<code>\hook_debug_on:</code>	18
		<code>\hook_disable_generic:n</code>	16
		<code>\hook_gclear_next_code:n</code>	17
		<code>\hook_gput_code:n:n</code>	17
		<code>\hook_gput_code_with_args:n:n</code> ...	17
		<code>\hook_gput_next_code:n</code>	17
		<code>\hook_gput_next_code_with_</code> <code>args:n</code>	17
		<code>\hook_gremove_code:n</code>	17
		<code>\hook_gset_rule:n:n</code>	18
		<code>\hook_if_empty:nTF</code>	18
		<code>\hook_if_empty_p:n</code>	18
		<code>\hook_log:n</code>	18
		<code>\hook_new:n</code>	16
		<code>\hook_new_pair:n</code>	15
		<code>\hook_new_pair_with_args:n</code> ...	16
		<code>\hook_new_pair_with_args:n:n</code> ...	16
		<code>\hook_new_reversed:n</code>	15
<code>\DebugHooksOff</code>	15		
<code>\DebugHooksOn</code>	15		
<code>\DebugMarksOff</code>	67		
<code>\DebugMarksOn</code>	67		
<code>\DebugShipoutsOff</code>	58		
<code>\DebugShipoutsOn</code>	58		
<code>\DebugSocketsOff</code>	87		
<code>\DebugSocketsOn</code>	87		
<code>\DeclareDefaultHookRule</code>	20		
<code>\DeclareHookRule</code>	7		
<code>\DeclareInstance</code>	95		
<code>\DeclareInstanceCopy</code>	96		
<code>\DeclareTemplateCode</code>	93		
<code>\DeclareTemplateCopy</code>	94		
<code>\DeclareTemplateInterface</code>	92		
dim commands:			
<code>\l_tmpa_dim</code>	92		
<code>\DisableGenericHook</code>	4		
<code>\DisableHook</code>	4		
<code>\DiscardShipoutBox</code>	57		
<code>\document</code>	28		
<code>\documentclass</code>	10		
		E	
<code>\edef</code>	68		
<code>\EditInstance</code>	97		
<code>\EditTemplateDefaults</code>	97		
<code>\end</code>	26		
<code>\endcsname</code>	8		
<code>enddocument</code> (hook)	24, 25, 29		
<code>\enddocument</code>	24		
<code>enddocument/afteraux</code> (hook)	29		
<code>enddocument/afterlastpage</code> (hook) ...	29		
<code>enddocument/end</code> (hook)	29		
<code>enddocument/info</code> (hook)	29		
<code>\endgraf</code>	48		
<code>env</code> (hook)	13, 22		
<code>env/<env>/after</code> (hook)	26, 26, 27		
<code>env/<env>/before</code> (hook)	26, 26, 27		
<code>env/<env>/begin</code> (hook)	26, 27		
<code>env/<env>/end</code> (hook)	26, 27		
<code>env/document/after</code> (hook)	30		
<code>env/document/before</code> (hook)	28		
<code>env/document/begin</code> (hook)	28		
<code>env/document/end</code> (hook)	30		
<code>env/quote/after</code> (hook)	20, 20		
<code>\ERROR</code>	50		
<code>\errorstopmode</code>	18		
<code>\everypar</code>	49		
<code>\EveryShipout</code>	60		

<code>\hook_new_reversed_with_args:nn</code>	16	<code>file/⟨package name⟩.sty/before</code>	34
<code>\hook_new_with_args:nn</code>	16	<code>file/after</code>	32, 33, 34
<code>\hook_show:n</code>	18	<code>file/array.sty/after</code>	33
<code>\hook_use:n</code>	21	<code>file/before</code>	32, 33, 34
<code>\hook_use:nnw</code>	16	<code>include</code>	22, 35
<code>\hook_use_once:n</code>	8	<code>include/⟨name⟩/after</code>	35
<code>\hook_use_once:nnw</code>	16	<code>include/⟨name⟩/before</code>	35
Hooks:		<code>include/⟨name⟩/end</code>	35
<code>/after</code>	32, 32, 34	<code>include/⟨name⟩/excluded</code>	35, 35
<code>/before</code>	32, 32	<code>include/after</code>	35
<code>A/foo</code>	13, 13	<code>include/before</code>	35
<code>after</code>	27	<code>include/end</code>	35
<code>babel/⟨language⟩/afterextras</code>	22	<code>include/excluded</code>	35, 35
<code>begindocument</code>	80, 80, 25, 25, 25, 28, 38	<code>insertmark</code>	31, 63
<code>begindocument/before</code>	28	<code>mdseries</code>	30
<code>begindocument/end</code>	28	<code>mdseries/defaults</code>	30
<code>bfseries</code>	30, 30	<code>myhook</code>	6
<code>bfseries/defaults</code>	30, 30	<code>normalfont</code>	30
<code>class</code>	22	<code>package</code>	22
<code>class/</code>	34	<code>package/</code>	34
<code>class/⟨name⟩/after</code>	34	<code>package/⟨name⟩/after</code>	34
<code>class/⟨name⟩/before</code>	34	<code>package/⟨name⟩/before</code>	34
<code>class/after</code>	34	<code>package/⟨package name⟩/after</code>	34
<code>class/before</code>	34	<code>package/⟨package name⟩/before</code>	34
<code>cmd</code>	16, 22, 22, 41, 42	<code>package/after</code>	34, 34
<code>cmd/⟨name⟩/after</code>	27, 38, 41	<code>package/before</code>	34, 34
<code>cmd/⟨name⟩/before</code>	27, 38, 38	<code>para/after</code>	47
<code>cmd/foo/before</code>	40	<code>para/before</code>	47
<code>cmd/include/after</code>	35	<code>para/begin</code>	47, 49
<code>cmd/include/before</code>	35	<code>para/end</code>	47, 48
<code>cmd/section/after</code>	41, 41	<code>rmfamily</code>	30, 30, 30
<code>cmd/section/before</code>	41	<code>selectfont</code>	31
<code>enddocument</code>	24, 25, 29	<code>sffamily</code>	30
<code>enddocument/afteraux</code>	29	<code>shipout</code>	55, 56, 56, 57, 57
<code>enddocument/afterlastpage</code>	29	<code>shipout/...</code>	54, 57
<code>enddocument/end</code>	29	<code>shipout/after</code>	
<code>enddocument/info</code>	29	54, 54, 54, 56, 56, 56, 58, 58
<code>env</code>	13, 22	<code>shipout/background</code>	
<code>env/⟨env⟩/after</code>	26, 26, 27	54, 55, 56, 57, 59, 59, 60
<code>env/⟨env⟩/before</code>	26, 26, 27	<code>shipout/before</code>	
<code>env/⟨env⟩/begin</code>	26, 27	...	80, 54, 54, 54, 54, 55, 55, 55,
<code>env/⟨env⟩/end</code>	26, 27	56, 56, 56, 57, 57, 57, 57, 58, 58, 60, 60
<code>env/document/after</code>	30	<code>shipout/firstpage</code>	
<code>env/document/before</code>	28	54, 54, 54, 55, 56, 56, 56
<code>env/document/begin</code>	28	<code>shipout/foreground</code>	
<code>env/document/end</code>	30	54, 55, 56, 57, 59, 59, 60
<code>env/quote/after</code>	20, 20	<code>shipout/lastpage</code>	
<code>expand@font@defaults</code>	30	29, 54, 54, 56, 56, 56, 56, 56
<code>file</code>	13, 22	<code>test</code>	23, 23
<code>file/⟨file name⟩/after</code>	33, 33	<code>toplevel</code>	25, 6, 6
<code>file/⟨file name⟩/before</code>	33, 33	<code>ttfamily</code>	30
<code>file/⟨filename⟩/after</code>	32		
<code>file/⟨filename⟩/before</code>	32		
<code>file/⟨package name⟩.sty/after</code>	34	I	
		<code>identity (plug)</code>	82, 85, 85, 85

<code>\IfHookEmptyF</code>	14		
<code>\IfHookEmptyT</code>	14		
<code>\IfHookEmptyTF</code>	14		
<code>\IfInstanceExistsF</code>	96		
<code>\IfInstanceExistsT</code>	96		
<code>\IfInstanceExistsTF</code>	96		
<code>\IfLabelExistsF</code>	76		
<code>\IfLabelExistsT</code>	76		
<code>\IfLabelExistsTF</code>	76		
<code>\IfMarksEqualF</code>	64		
<code>\IfMarksEqualT</code>	64		
<code>\IfMarksEqualTF</code>	64		
<code>\ifnum</code>	58		
<code>\IfPropertyExistsF</code>	76		
<code>\IfPropertyExistsT</code>	76		
<code>\IfPropertyExistsTF</code>	76		
<code>\IfPropertyRecordedF</code>	76		
<code>\IfPropertyRecordedT</code>	76		
<code>\IfPropertyRecordedTF</code>	76		
<code>\IfSocketExistsTF</code>	86		
<code>\IfSocketPlugAssignedTF</code>	86		
<code>\IfSocketPlugExistsTF</code>	86		
<code>\ignorespaces</code>	28		
<code>\immediate</code>	29		
<code>include (hook)</code>	22, 35		
<code>\include</code>	35		
<code>include/.../after</code>	35		
<code>include/.../before</code>	35		
<code>include/.../end</code>	35		
<code>include/<name>/after (hook)</code>	35		
<code>include/<name>/before (hook)</code>	35		
<code>include/<name>/end (hook)</code>	35		
<code>include/<name>/excluded (hook)</code> ..	35, 35		
<code>include/after (hook)</code>	35		
<code>include/after</code>	35		
<code>include/before (hook)</code>	35		
<code>include/before</code>	35		
<code>include/end (hook)</code>	35		
<code>include/end</code>	35		
<code>include/excluded (hook)</code>	35, 35		
<code>\includeonly</code>	35		
<code>\indent</code>	49		
<code>\IndentBox</code>	49		
<code>\index</code>	63		
<code>\input</code>	33		
<code>\insert</code>	45		
<code>\InsertMark</code>	31		
<code>insertmark (hook)</code>	31, 63		
<code>insertmark</code>	63		
<code>\InstanceValue</code>	95		
int commands:			
<code>\int_use:N</code>	58		
<code>\l_tmpa_int</code>	92		
<code>\item</code>	44		
		K	
		<code>\KeyValue</code>	91
		L	
		<code>\label</code>	76
		<code>label</code>	77
		<code>\lastbox</code>	52
		<code>\LastMark</code>	64
		<code>\leftmark</code>	62
		<code>\listfiles</code>	29
		<code>\LoadPackageWithOptions</code>	33
		<code>\LogHook</code>	14
		<code>\LogSocket</code>	83
		M	
		<code>\marginpar</code>	47
		<code>\mark</code>	62
		mark commands:	
		<code>\mark_clear_structure:n</code>	71
		<code>\mark_copy_structure:nn</code>	70
		<code>\mark_debug_off:</code>	67
		<code>\mark_debug_on:</code>	67
		<code>\mark_get_marks_for_reinsertion:nnN</code>	71
		<code>\mark_if_eq:nnnnnnTF</code>	64
		<code>\mark_if_eq:nnnnTF</code>	64
		<code>\mark_insert:nn</code>	71
		<code>\mark_new_class:n</code>	63
		<code>\mark_set_structure_to_err:n</code> ...	70
		<code>\mark_update_structure_from_- material:nn</code>	70
		<code>\mark_use_first:nn</code>	64
		<code>\mark_use_last:nn</code>	64
		<code>\mark_use_top:nn</code>	64
		mark internal commands:	
		<code>__mark_update_dblcol_structures:</code>	70
		<code>__mark_update_dblcol_structures:</code>	71
		<code>__mark_update_singlecol_- structures:</code>	70
		<code>\markboth</code>	62
		<code>\markright</code>	62
		<code>mdseries (hook)</code>	30
		<code>\mdseries</code>	30
		<code>mdseries/defaults (hook)</code>	30
		muskip commands:	
		<code>\l_tmpa_muskip</code>	92
		<code>myhook (hook)</code>	6
		N	
		<code>\newcommand</code>	39
		<code>\NewCommandCopy</code>	41
		<code>\NewDocumentCommand</code>	42

<code>\@kernel@before@para@before</code>	48	<code>\TopMark</code>	68
<code>\@kernel@before@para@begin</code>	47	<code>\topmark</code>	62
<code>\@opcol</code>	71	<code>totalpages</code>	58
<code>\@outputbox</code>	71	<code>ttfamily (hook)</code>	30
<code>\@@end</code>	29	<code>\ttfamily</code>	30
<code>\AddToHook</code>	38	<code>\typeout</code>	56
<code>\AddToHookNext</code>	38		
<code>\AddToHookNextWithArguments</code>	38	U	
<code>\AddToHookWithArguments</code>	38	<code>\unitlength</code>	55
<code>\apptocmd</code>	39	<code>\unpenalty</code>	52
<code>\botmark</code>	69	<code>\unskip</code>	52
<code>\declare@file@substitution</code>	36	<code>\UnusedTemplateKeys</code>	93
<code>\DeclareRobustCommand</code>	40	<code>\unvcopy</code>	70
<code>\def</code>	40	use commands:	
<code>\disable@package@load</code>	36	<code>\use_i:nn</code>	92
<code>\expand@font@defaults</code>	30	<code>\UseHook</code>	21
<code>\if@firstcolumn</code>	71	<code>\UseHookWithArguments</code>	41
<code>\input@path</code>	33	<code>\UseInstance</code>	96
<code>\new@label@record</code>	75	<code>\UseOneTimeHook</code>	34
<code>\newcommand</code>	40	<code>\UseOneTimeHookWithArguments</code>	5
<code>\NewDocumentCommand</code>	40	<code>\usepackage</code>	32
<code>\on@line</code>	50	<code>\UseSocket</code>	86
<code>\patchcmd</code>	40	<code>\UseTemplate</code>	96
<code>\pretocmd</code>	39	<code>\usetikzlibrary</code>	10
<code>\protected@edef</code>	68		
<code>\reenable@package@load</code>	36	V	
<code>\section</code>	41	<code>\vadjust</code>	45
<code>\topmark</code>	69	<code>\vbox</code>	46
<code>\topmark(s)</code>	69	<code>\vskip</code>	45
<code>\undeclare@file@substitution</code>	36		
<code>\unexpanded</code>	64	W	
tex commands:		<code>\write</code>	56
<code>\tex_savepos:D</code>	73		
<code>\thepage</code>	77	X	
<code>title</code>	77	<code>xpos</code>	77
tl commands:			
<code>\l_tmpa_tl</code>	92	Y	
<code>toplevel (hook)</code>	25, 6, 6	<code>ypos</code>	77