

The `lthooks` package*

Frank Mittelbach†

July 21, 2020

Contents

1	Introduction	1
2	Package writer interface	2
2.1	L ^A T _E X 2 _ε interfaces	2
2.1.1	Declaring hooks and using them in code	2
2.1.2	Updating code for hooks	3
2.1.3	Hook names and default labels	4
2.1.4	Defining relations between hook code	5
2.1.5	Querying hooks	7
2.1.6	Displaying hook code	7
2.1.7	Debugging hook code	8
2.2	L ³ programming layer (<code>exp13</code>) interfaces	8
2.3	On the order of hook code execution	10
2.4	The use of “reversed” hooks	11
2.5	Private L ^A T _E X kernel hooks	12
2.6	Legacy L ^A T _E X 2 _ε interfaces	13
2.7	L ^A T _E X 2 _ε commands and environments augmented by hooks	13
2.7.1	Generic hooks for all environments	13
2.7.2	Hooks provided by <code>\begin{document}</code>	14
2.7.3	Hooks provided by <code>\end{document}</code>	15
2.7.4	Hooks provided <code>\shipout</code> operations	16
2.7.5	Hooks provided file loading operations	16
	Index	16

*This package has version v0.9b dated 2020/07/19, © L^AT_EX Project.

†Code improvements for speed and other goodies by Phelype Oleinik

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.

2 Package writer interface

The hook management system is offered as a set of CamelCase commands for traditional $\text{\LaTeX} 2_{\epsilon}$ 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 \LaTeX . 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.

2.1 $\text{\LaTeX} 2_{\epsilon}$ interfaces

2.1.1 Declaring hooks and using them in code

With two exceptions, hooks have to be declared before they can be used. The exceptions are hooks in environments (i.e., executed at `\begin` and `\end`) and hooks run when loading files, e.g. before and after a package is loaded, etc. Their hook names depend on the environment or the file name and so declaring them beforehand is difficult.

`\NewHook`

`\NewHook {<hook>}`

Creates a new `<hook>`. If this is a hook provided as part of a package it is suggested that the `<hook>` 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 [2.1.3](#).

`\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 [2.3](#) and [2.4](#) for further details.

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section [2.1.3](#).

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

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

The `\hooks` can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

`\UseHook` `{\hook}`

Execute the hook code inside a command or environment.¹

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 `\hook` cannot be specified using the dot-syntax. A leading `.` is treated literally.

`\UseOneTimeHook` `{\hook}`

Some hooks are only used (and can be only used) in one place, for example, those in `\begin{document}` or `\end{document}`. Once we have passed that point adding to the hook through a defined `\addto-cmd` 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 `\addto-cmd` 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.

FMi: Maybe add an error version as well?

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

2.1.2 Updating code for hooks

`\AddToHook` `{\hook}[\label]{\code}`

Adds `\code` to the `\hook` labeled by `\label`. If the optional argument `\label` is not provided, if `\AddToHook` is used in a package/class, then the current package/class name is used, otherwise `top-level` is used (see section 2.1.3).

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 later 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 hook of other packages without worrying whether they are actually used in the current document. See section 2.1.5.

The `\hook` and `\label` can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

\RemoveFromHook

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

Removes any code labeled by `<label>` from the `<hook>`. If the optional argument `<label>` is not provided, if `\AddToHook` is used in a package/class, then the current package/class name is used, otherwise `top-level` is used.

If the optional argument is `*`, then all code chunks are removed. This is rather dangerous as it drops code from other packages one may not know about!

The `<hook>` and `<label>` can be specified using the dot-syntax to denote the current package name. See section [2.1.3](#).

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

Note that you can't cancel the setting with

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

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 to 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 the 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 happens after the 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

The 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 [2.1.5](#).

The `<hook>` can be specified using the dot-syntax to denote the current package name. See section [2.1.3](#).

2.1.3 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>`.

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

Using an explicit $\langle label \rangle$ 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 $\langle label \rangle$ throughout the sub-packages in order to avoid that the labels change if you internally reorganize your code.

It is not enforced, but highly recommended that the hooks defined by a package, and the $\langle labels \rangle$ 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 $\langle hook \rangle$ and in a $\langle label \rangle$. If $\langle hook \rangle$ name or $\langle label \rangle$ consist just of a single dot (\cdot), or starts with a dot followed by a slash ($\cdot/$) then the dot denotes the $\langle default label \rangle$ (usually the current package or class name—see `\DeclareDefaultHookLabel`). A “ \cdot ” or “ $\cdot/$ ” anywhere else in a $\langle hook \rangle$ or in $\langle label \rangle$ 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}{.}{<}{babel}
\AddToHook  {file/after/foo.tex}{code}
```

are equivalent to:

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

The $\langle default label \rangle$ is automatically set to the name of the current package or class (using `\@currname`). If `\@currname` is not set (because 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 $\langle default label \rangle$.

This syntax is available in all $\langle label \rangle$ arguments and most $\langle hook \rangle$, both in the $\text{\LaTeX} 2_{\epsilon}$ interface, and the $\text{\LaTeX} 3$ interface described in section 2.2.

Note, however, that the replacement of \cdot by the $\langle default label \rangle$ takes place when the hook command is executed, so actions that are somehow executed after the package ends will have the wrong $\langle default label \rangle$ 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`) and `\IfHookExistTF` (and `\hook_if_exist: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 it in logical parts, but still use the main package name as $\langle label \rangle$, then the $\langle default label \rangle$ can be set using `\DeclareDefaultHookLabel`:

`\DeclareDefaultHookLabel` `\DeclareDefaultHookLabel` $\langle\textit{default label}\rangle$

Sets the $\langle\textit{default label}\rangle$ to be used in $\langle\textit{label}\rangle$ arguments. If `\DeclareDefaultHookLabel` is not used in the current package, `\@currname` is used instead. If `\@currname` is not set, the code is assumed to be in the main document, in which case `top-level` is used.

The effect of `\DeclareDefaultHookLabel` holds for the current file, and is reset to the previous value when the file is closed.

2.1.4 Defining relations between hook code

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

Before the hook management system was introduced packages had to take elaborate precaution to determine if some other package got loaded as well (before or after) and 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 code added to hooks got added in the right order and 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 explicitly describe in which order they should be processed.

`\DeclareHookRule` `\DeclareHookRule` $\langle\textit{hook}\rangle$ $\langle\textit{label1}\rangle$ $\langle\textit{relation}\rangle$ $\langle\textit{label2}\rangle$

Defines a relation between $\langle\textit{label1}\rangle$ and $\langle\textit{label2}\rangle$ for a given $\langle\textit{hook}\rangle$. If $\langle\textit{hook}\rangle$ is `??` this defines a relation for all hooks that use the two labels, i.e., that have chunks of code labeled with $\langle\textit{label1}\rangle$ and $\langle\textit{label2}\rangle$. Rules specific to a given hook take precedence over default rules that use `??` as the $\langle\textit{hook}\rangle$.

Currently, the supported relations are the following:

before or **<** Code for $\langle\textit{label1}\rangle$ comes before code for $\langle\textit{label2}\rangle$.

after or **>** Code for $\langle\textit{label1}\rangle$ comes after code for $\langle\textit{label2}\rangle$.

incompatible-warning Only code for either $\langle\textit{label1}\rangle$ or $\langle\textit{label2}\rangle$ 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 \LaTeX error is raised, and the code for both labels are dropped from that hook until the conflict is resolved.

removes Code for $\langle\textit{label1}\rangle$ overwrites code for $\langle\textit{label2}\rangle$. More precisely, code for $\langle\textit{label2}\rangle$ 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 $\langle\textit{label1}\rangle$ and $\langle\textit{label2}\rangle$ is irrelevant. This rule is there to undo an incorrect rule specified earlier.

The $\langle\textit{hook}\rangle$ and $\langle\textit{label}\rangle$ can be specified using the dot-syntax to denote the current package name. See section [2.1.3](#).

`\ClearHookRule` `\ClearHookRule{<hook>}{<label1>}{<label2>}`

Syntactic sugar for saying that `<label1>` and `<label2>` are unrelated for the given `<hook>`.

`\DeclareDefaultHookRule` `\DeclareDefaultHookRule{<label1>}{<relation>}{<label2>}`

This sets up a relation between `<label1>` and `<label2>` 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.)

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

The `<label>` can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

2.1.5 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: they have four possible states. A hook may exist or not, and either way it may or may not be empty. This means that even a hook that doesn't exist may be non-empty.

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.

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.

`\IfHookEmptyTF` ★ `\IfHookEmptyTF {<hook>} {<true code>} {<>false code>}`

Tests if the `<hook>` is empty (*i.e.*, no code was added to it using either `\AddToHook` or `\AddToHookNext`), and branches to either `<true code>` or `<>false code>` depending on the result.

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

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

`\IfHookExistTF` ★ `\IfHookExistTF {<hook>} {<true code>} {<false code>}`

Tests if the *<hook>* exists (if it was created with either `\NewHook`, `\NewReversedHook`, or `\NewMirroredHookPair`), and branches to either *<true code>* or *<false code>* depending on the result.

The existence of a hook usually doesn't mean much from the viewpoint of code that tries to add/remove code from that hook, since package loading order may vary, thus the creation of hooks is asynchronous to adding and removing code from it, so this test should be used sparingly.

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

FMi: Would be helpful if we provide some use cases

2.1.6 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.

`\ShowHook` `\ShowHook {<hook>}`

Displays information about the *<hook>* such as

- the code chunks (and their labels) added to it,
- any rules set up to order them,

FMi: currently this is missing the default rules that apply, guess that needs fixing

- the computed order (if already defined),
- any code executed on the next invocation only.

The *<hook>* can be specified using the dot-syntax to denote the current package name. See section [2.1.3](#).

2.1.7 Debugging hook code

`\DebugHookOn` `\DebugHookOn`

`\DebugHookOff`

Turn the debugging of hook code on or off. This displays changes made to the hook data structures. The output is rather coarse and not really intended for normal use.

2.2 L3 programming layer (expl3) interfaces

This is a quick summary of the $\text{\LaTeX}3$ programming interfaces for use with packages written in `expl3`. In contrast to the $\text{\LaTeX}2_{\epsilon}$ interfaces they always use mandatory arguments only, e.g., you always have to specify the *<label>* 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.

<code>\hook_new:n</code>	<code>\hook_new:n{<hook>}</code>
<code>\hook_new_reversed:n</code>	<code>\hook_new_pair:nn{<hook-1>}{<hook-2>}</code>
<code>\hook_new_pair:nn</code>	

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 $\{ \langle hook-2 \rangle \}$ 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 2.1.3.

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

Executes the $\{ \langle hook \rangle \}$ code followed (if set up) by the code for next invocation only, then empties that next invocation code.

The $\langle hook \rangle$ *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>
-------------------------------	--

Changes the $\{ \langle hook \rangle \}$ status so that from now on any addition to the hook code is executed immediately. Then execute any $\{ \langle hook \rangle \}$ code already set up.

FMi: better L3 name?

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

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

Adds a chunk of $\langle code \rangle$ to the $\langle hook \rangle$ labeled $\langle label \rangle$. If the label already exists the $\langle code \rangle$ is appended to the already existing code.

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

The $\langle hook \rangle$ and $\langle label \rangle$ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

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

Adds a chunk of $\langle code \rangle$ for use only in the next invocation of the $\langle hook \rangle$. Once used it is gone.

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 $\langle code \rangle$.

The $\langle hook \rangle$ can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

`\hook_gremove_code:nm` `\hook_gremove_code:nm {<hook>} {<label>}`

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

If the code for that `<label>` wasn't yet added to the `<hook>`, an order is set so that when some code attempts to add that label, the removal order takes action and the code is not added.

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 [2.1.3](#).

`\hook_gset_rule:nmnn` `\hook_gset_rule:nmnn {<hook>} {<label1>} {<relation>} {<label2>}`

Relate `<label1>` with `<label2>` when used in `<hook>`. See `\DeclareHookRule` for the allowed `<relation>`s. If `<hook>` is `??` a default rule is specified.

The `<hook>` and `<label>` can be specified using the dot-syntax to denote the current package name. See section [2.1.3](#). The dot-syntax is parsed in both `<label>` arguments, but it usually makes sense to be used in only one of them.

`\hook_if_empty_p:n *` `\hook_if_empty:n {<hook>} {<>true code>} {<>false code>}`
`\hook_if_empty:nTF *`

Tests if the `<hook>` is empty (*i.e.*, no code was added to it using either `\AddToHook` or `\AddToHookNext`), and branches to either `<>true code>` or `<>false code>` depending on the result.

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

`\hook_if_exist_p:n *` `\hook_if_exist:n {<hook>} {<>true code>} {<>false code>}`
`\hook_if_exist:nTF *`

Tests if the `<hook>` exists (if it was created with either `\NewHook`, `\NewReversedHook`, or `\NewMirroredHookPair`), and branches to either `<>true code>` or `<>false code>` depending on the result.

FMi: what are the results for generic hooks that do not need to be declared?

The existence of a hook usually doesn't mean much from the viewpoint of code that tries to add/remove code from that hook, since package loading order may vary, thus the creation of hooks is asynchronous to adding and removing code from it, so this test should be used sparingly.

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

`\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.

2.3 On the order of hook code execution

Chunks of code for a `<hook>` 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}
Extra code 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, or 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}
Extra code next invocation:
---
Rules:
  packageA|packageB with relation before
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}{label-3}
```

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).

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 the user wants the quotes also in blue and therefore adds:

```
\usepackage{color}
\AddToHook{env/quote/before}{\begin{color}{blue}}
\AddToHook{env/quote/after} {\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, top-level
```

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

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

and an error message that `\begin{color}` 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}
  top-level -> \end {color}
Extra code next invocation:
---
Rules:
---
Execution order (after reversal):
  top-level, package-1
```

The reversal of the execution order happens before applying any rules, 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.

2.5 Private L^AT_EX kernel hooks

There are a few places where it is absolutely essential for L^AT_EX 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 L^AT_EX).

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@{hookname}` or `\@kernel@after@{hookname}`. 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.⁵

2.6 Legacy L^AT_EX 2_ε interfaces

L^AT_EX 2_ε 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 L^AT_EX and more will follow. See the next section for what is already available.

<code>\AtBeginDocument</code>	<code>\AtBeginDocument [<i>label</i>] {<i>code</i>}</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` if done outside of a package or class or with the package/class name if called inside such a file.

This way one can add further 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 the `top-level` code. When using the optional argument the call is equivalent to running `\AddToHook {begindocument} [label] {code}`.

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.

<code>\AtEndDocument</code>	<code>\AtEndDocument [<i>label</i>] {<i>code</i>}</code>
-----------------------------	--

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

<code>\AtBeginDvi</code>	<code>\AtBeginDvi [<i>label</i>] {<i>code</i>}</code>
--------------------------	---

This hook is discussed in conjunction with the shipout hooks.

⁵As with everything in T_EX 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!”

2.7 L^AT_EX 2_ε commands and environments augmented by hooks

intro to be written

2.7.1 Generic hooks for all environments

Every environment `<env>` has now four associated hooks coming with it:

env/<env>/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/<env>/begin This hook is executed as part of `\begin` directly in front of the code specific to the environment start (e.g., the second argument of `\newenvironment`). Its scope is the environment body.

env/<env>/end This is executed as part of `\end` directly in front of the code specific to the end of the environment (e.g., the third argument of `\newenvironment`).

env/<env>/after This 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.

This hook is implemented as a reversed hook so if two packages add code to `env/<env>/before` and to `env/<env>/after` they can add surrounding environments and the order of closing them happens in the right sequence.

In contrast to other hooks these hooks do not need to be declared using `\NewHook`.

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 `\end<env>` (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.

2.7.2 Hooks provided by `\begin{document}`

Until 2020 `\begin{document}` offered exactly one hook that one had to fill using `\AtBeginDocument`. Experience has 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 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:

env/document/before This is the generic environment hook executed effectively before `\begin{document}` starts, i.e., one can think of it as a hook for code at the end of the preamble section.

env/document/begin This is the second generic environment hook that is executed after the environment has started its group. But given that for the `document` environment this group is canceled there is little difference to the previous one as the two are directly executed one after another (the only difference is that in this hook `\@currentenv` is now set to `document` but anybody adding to this hook would know that already).

begindocument This hook is added to by `\AtBeginDocument` and is executed after the `.aux` file as be read in 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.

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`.

2.7.3 Hooks provided by `\end{document}`

L^AT_εX 2_ε always provided `\AtEndDocument` to add code to the execution of `\end{document}` just in front of the code that is normally executed there. While this was a big improvement over the situation in L^AT_εX 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:

env/document/end The generic hook inside `\end`.

enddocument The hook associated with `\AtEndDocument`. It is immediately called after the previous hook so there could be just one.⁶

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.

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). 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.).

⁶We could make `\AtEndDocument` just fill the `env/document/end` but maybe that is a bit confusing.

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`.

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.

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

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, `LATEX` needs to be run several times, so initially it might get executed on the wrong page. See section 2.7.4 for where to find the details.

2.7.4 Hooks provided `\shipout` operations

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

2.7.5 Hooks provided 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`, etc. These are documented in `ltfilehook-doc.pdf` or with code in `ltfilehook-code.pdf`.

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	B
<code>\<addto-cmd></code> 2	<code>\begin</code> 1, 13, 14
<code>\<env></code> 14	C
A	<code>\ClearHookRule</code> 6
<code>\AddToHook</code> 1, 2, 3, 3, 4, 7, 9, 13	<code>\clearpage</code> 15
<code>\AddToHookNext</code> 3, 4, 7, 9	D
<code>\AtBeginDocument</code> 2, 13, 13, 14	<code>\DebugHookOff</code> 8
<code>\AtBeginDvi</code> 13	<code>\DebugHookOn</code> 8
<code>\AtEndDocument</code> 13, 15	<code>\DeclareDefaultHookLabel</code> 4, 5, 5
	<code>\DeclareDefaultHookRule</code> 6

<code>\DeclareHookRule</code>	1, 6, 6, 9	<code>\input</code>	16
<code>\document</code>	14		
<code>\documentclass</code>	5		
	E		
<code>\end</code>	1, 13-15		
<code>\end(env)</code>	14		
<code>\enddocument</code>	12, 15		
	H		
hook commands:			
<code>\hook_debug_off:</code>	10		
<code>\hook_debug_on:</code>	10		
<code>\hook_gput_code:nnn</code>	9, 9		
<code>\hook_gput_next_code:nn</code>	9		
<code>\hook_gremove_code:nn</code>	9		
<code>\hook_gset_rule:nnnn</code>	9		
<code>\hook_if_empty:n</code>	9		
<code>\hook_if_empty:nTF</code>	5, 9		
<code>\hook_if_empty_p:n</code>	9		
<code>\hook_if_exist:n</code>	10		
<code>\hook_if_exist:nTF</code>	5, 10		
<code>\hook_if_exist_p:n</code>	10		
<code>\hook_new:n</code>	8		
<code>\hook_new_pair:nn</code>	8		
<code>\hook_new_reversed:n</code>	8		
<code>\hook_use:n</code>	5, 8		
<code>\hook_use_once:n</code>	8		
	I		
<code>\IfHookEmptyTF</code>	5, 7		
<code>\IfHookExistTF</code>	5, 7		
<code>\ignorespaces</code>	14		
<code>\include</code>	16		
	L		
<code>\listfiles</code>	15		
	N		
<code>\newenvironment</code>	13		
<code>\NewHook</code>	2, 2, 2, 7, 7, 10, 11, 14		
<code>\NewMirroredHookPair</code>	2, 7, 10		
<code>\NewReversedHook</code>	2, 2, 7, 10, 11		
<code>\normalsize</code>	3		
	R		
<code>\RemoveFromHook</code>	3, 3		
	S		
<code>\shipout</code>	16		
<code>\ShowHook</code>	8, 10, 12		
<code>\small</code>	3		
<code>\special</code>	16		
	T		
TeX and L ^A T _E X 2 _ε commands:			
<code>\@currentenv</code>	14		
<code>\@currname</code>	5, 5		
<code>\@firstofone</code>	2		
<code>\@kernel@after@hookname</code>	12		
<code>\@kernel@before@hookname</code>	12		
<code>\@@end</code>	15		
	U		
<code>\UseHook</code>	2, 5, 10		
<code>\UseOneTimeHook</code>	2		
<code>\usepackage</code>	5, 16		