The \texttt{lthooks} package\footnote{This package has version v1.0n dated 2021/05/26, \copyright{} \LaTeX{} Project.}

Frank Mittelbach\footnote{Code improvements for speed and other goodies by Phelype Oleinik}

June 15, 2021

Contents

1 Introduction 2

2 Package writer interface 2

2.1 \LaTeX{}\,2\epsilon interfaces .................................. 2

2.1.1 Declaring hooks .......................................... 2

2.1.2 Special declarations for hooks ............................. 3

2.1.3 Using hooks in code ...................................... 3

2.1.4 Updating code for hooks .................................. 4

2.1.5 Hook names and default labels ............................. 5

2.1.6 The \texttt{top-level} label ................................. 7

2.1.7 Defining relations between hook code ....................... 8

2.1.8 Querying hooks ......................................... 9

2.1.9 Displaying hook code ................................... 10

2.1.10 Debugging hook code .................................. 11

2.2 L3 programming layer (\texttt{expl3}) interfaces ............. 12

2.3 On the order of hook code execution .......................... 14

2.4 The use of “reversed” hooks ................................ 15

2.5 Difference between “normal” and “one-time” hooks .......... 16

2.6 Private \LaTeX{} kernel hooks .................................. 17

2.7 Legacy \LaTeX{}\,2\epsilon interfaces .......................... 18

2.8 \LaTeX{}\,2\epsilon commands and environments augmented by hooks 18

2.8.1 Generic hooks for all environments ....................... 18

2.8.2 Generic hooks for commands ................................ 20

2.8.3 Generic hooks provided by file loading operations ........ 20

2.8.4 Hooks provided by \texttt{\begin{document}} .................. 20

2.8.5 Hooks provided by \texttt{\end{document}} .................. 21

2.8.6 Hooks provided by \texttt{\shipout} operations ............... 22

2.8.7 Hooks provided in NFSS commands .......................... 22

Index 23
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 \LaTeXe 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 \LaTeXe interfaces

2.1.1 Declaring hooks

With a few exceptions, hooks have to be declared before they can be used. The exceptions are the generic hooks for commands, 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 command, environment or the file name and so declaring them beforehand is not practical.

\begin{itemize}
\item `\NewHook{⟨hook⟩}`
\end{itemize}

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 2.1.5.

\begin{itemize}
\item `\NewReversedHook{⟨hook⟩}`
\end{itemize}

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 2.1.5.
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 2.1.5.

### 2.1.2 Special declarations for hooks

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

`\DisableHook{⟨hook⟩}`

After this declaration the ⟨hook⟩ is no longer usable: Any attempt to add further 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 `ltcmdhooks-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 ⟨hook⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

`\ProvideHook{⟨hook⟩}`

Like `\NewHook` but does nothing if the hook was previously declared with `\NewHook`. This declaration should only be used in special situations, e.g., when command of another package need to be altered and it is not clear if for that command a generic hook was already explicitly declared before.

Normally `\NewHook` should be used instead.

`\ProvideReversedHook{⟨hook⟩}`

Like `\NewReversedHook` but does nothing if the hook was previously declared as a reversed hook.

`\ProvideMirroredHookPair{⟨hook-1⟩}{⟨hook-2⟩}`

A shorthand for `\ProvideHook{⟨hook-1⟩}\ProvideReversedHook{⟨hook-2⟩}`.

### 2.1.3 Using hooks in code

`\UseHook{⟨hook⟩}`

Execute the hook code inside a command or environment.

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.
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 \(\texttt{\textbackslash addto-cmd}\) command (e.g., \texttt{\textbackslash AddToHook} or \texttt{\textbackslash 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 \(\texttt{\textbackslash addto-cmd}\) to simply process its argument, i.e., essentially make it behave like \texttt{\@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 \(\texttt{\textbackslash hook}\) cannot be specified using the dot-syntax. A leading . is treated literally. See section 2.1.5 for details.

### 2.1.4 Updating code for hooks

\AddToHook \AddToHook \{\texttt{\textbackslash hook}\}\{\{\texttt{\textbackslash label}\}\}\{\{\texttt{\textbackslash code}\}\}

Adds \(\texttt{\textbackslash code}\) to the \(\texttt{\textbackslash hook}\) labeled by \(\texttt{\textbackslash label}\). When the optional argument \(\texttt{\textbackslash label}\) is not provided, the \(\texttt{\textbackslash default label}\) is used (see section 2.1.5). If \texttt{\textbackslash AddToHook} is used in a package/class, the \(\texttt{\textbackslash default label}\) is the package/class name, otherwise it is \texttt{\textbackslash top-level} (the \texttt{\textbackslash top-level} label is treated differently: see section 2.1.6).

If there already exists code under the \(\texttt{\textbackslash label}\) then the new \(\texttt{\textbackslash code}\) is appended to the existing one (even if this is a reversed hook). If you want to replace existing code under the \(\texttt{\textbackslash label}\), first apply \texttt{\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 \(\texttt{\textbackslash code}\) 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 2.1.8.

The \(\texttt{\textbackslash hook}\) and \(\texttt{\textbackslash label}\) can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

\RemoveFromHook \RemoveFromHook \{\texttt{\textbackslash hook}\}\{\{\texttt{\textbackslash label}\}\}

Removes any code labeled by \(\texttt{\textbackslash label}\) from the \(\texttt{\textbackslash hook}\). When the optional argument \(\texttt{\textbackslash label}\) is not provided, the \(\texttt{\textbackslash default label}\) is used (see section 2.1.5).

If the code for that \(\texttt{\textbackslash label}\) wasn’t yet added to the \(\texttt{\textbackslash hook}\), 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 optional \(\texttt{\textbackslash label}\) argument is *, then all code chunks are removed. This is rather dangerous as it drops code from other packages one may not know about and should therefore not by used by packages but only in document preambles!

The \(\texttt{\textbackslash hook}\) and \(\texttt{\textbackslash label}\) can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

In contrast to the voids relationship between two labels in a \texttt{\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/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 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{⟨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.\footnote{There is no mechanism to reorder such code chunks (or delete them).}

It is possible to nest declarations 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 2.1.8.

The ⟨hook⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

2.1.5 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/after/foo.tex}{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/after/foo.tex}{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 \LaTeX2ε interface, and the \LaTeX3 interface described in section 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 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 \SetDefaultHookLabel or \PushDefaultHookLabel{..}...\PopDefaultHookLabel.

\PushDefaultHookLabel \{\langle default label\rangle\}
\PopDefaultHookLabel

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

Inside a package or class, the \langle default label\rangle is equal to the package or class name, unless explicitly changed. Everywhere else, the \langle default label\rangle is top-level (see section 2.1.6) unless explicitly changed.

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

\PushDefaultHookLabel\{\langle package name\rangle\}
\PopDefaultHookLabel

to set the \langle default label\rangle for the package or class file. Inside the \langle package code\rangle the \langle default label\rangle can also be changed with \SetDefaultHookLabel. \input and other file input-related commands from the \LaTeX{} kernel do not use \PushDefaultHookLabel, so code within files loaded by these commands does not get a dedicated \langle label\rangle! (that is, the \langle default label\rangle 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 emulate \usepackage-like hook behaviour 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 \langle default label\rangle to top-level.

\SetDefaultHookLabel \{\langle default label\rangle\}

Similarly to \PushDefaultHookLabel, sets the current \langle default label\rangle to be used in \langle label\rangle 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 \langle default label\rangle.

This command is useful when a large package is composed of several smaller packages, but all should have the same \langle label\rangle, 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 \langle default label\rangle is top-level and there is no \PopDefaultHookLabel to end its effect. It is also not allowed to change the \langle default label\rangle to top-level.

2.1.6 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.

### 2.1.7 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 precaution to determine of some other package got loaded as well (before or after) and find some ways to alter its behavior accordingly. In addition is 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.
\DefineHookRule{⟨hook⟩}{⟨label1⟩}{⟨relation⟩}{⟨label2⟩}

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

Currently, the supported relations are the following:

before or < Code for ⟨label1⟩ comes before code for ⟨label2⟩.

after or > Code for ⟨label1⟩ comes after code for ⟨label2⟩.

\textbf{incompatible-warning} Only code for either ⟨label1⟩ or ⟨label2⟩ 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.

\textbf{incompatible-error} Like \textbf{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.

\textbf{voids} Code for ⟨label1⟩ overwrites code for ⟨label2⟩. More precisely, code for ⟨label2⟩ 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.

\textbf{unrelated} The order of code for ⟨label1⟩ and ⟨label2⟩ 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 \DefineHookRule overwrites any previous declaration.

The ⟨hook⟩ and ⟨label⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

\ClearHookRule{⟨hook⟩}{⟨label1⟩}{⟨label2⟩}

Syntactic sugar for saying that ⟨label1⟩ and ⟨label2⟩ are unrelated for the given ⟨hook⟩.

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

The ⟨label⟩ can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

\subsection{Querying hooks}

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

\begin{itemize}
  \item exist and be empty;
\end{itemize}
• 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 either way 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 \texttt{A} defines hook
\texttt{A/foo}, and package \texttt{B} adds some code to that hook. However, a document may load
package \texttt{B} before package \texttt{A}, or may not load package \texttt{A} at all. In both cases some code
is added to hook \texttt{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 \texttt{NewHook}
or some variant thereof. Generic hooks such as \texttt{file} and \texttt{env} hooks are automatically
declared when code is added to them.

\section{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.

\begin{verbatim}
\IfHookEmptyTF \IfHookEmptyTF {(hook)} {(true code)} {(false code)}
Tests if the \texttt{(hook)} is empty (\textit{i.e.}, no code was added to it using either \texttt{AddToHook}
or \texttt{AddToHookNext}) or such code was removed again (via \texttt{RemoveFromHook}), and branches
to either \texttt{(true code)} or \texttt{(false code)} depending on the result.

The \texttt{(hook)} cannot be specified using the dot-syntax. A leading \texttt{.} is treated literally.
\end{verbatim}

\begin{verbatim}
\ShowHook \ShowHook {(hook)}
Displays information about the \texttt{(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 \texttt{.log} file, and \ShowHook prints them to the
terminal/command window and starts \TeX’s prompt (only in \texttt{errorstopmode}) to wait
for user action.

The \texttt{(hook)} can be specified using the dot-syntax to denote the current package
name. See section \ref{sections}.
\end{verbatim}

10
Suppose a hook `example-hook` whose output of `\ShowHook{example-hook}` is:

```
-> The hook 'example-hook':
 > Code chunks:
   > foo -> [code from package 'foo']
   > bar -> [from package 'bar']
   > baz -> [package 'baz' is here]
 > Document-level (top-level) code (executed last):
   -> [code from 'top-level']
 > Extra code for next invocation:
   -> [one-time code]
 > Rules:
   > foo|baz with relation >
   > baz|bar with default relation <
 > Execution order (after applying rules):
   > 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 (first|last)):
-> ⟨top-level code⟩
```

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

```
-> ⟨next-code⟩
```

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

```
⟨label-1⟩|⟨label-2⟩ with (default?) relation ⟨relation⟩
```

which means that the ⟨relation⟩ applies to ⟨label-1⟩ and ⟨label-2⟩, in that order, as detailed in `\DeclareHookRule`. If the relation is `default` it means that that rule applies to ⟨label-1⟩ and ⟨label-2⟩ 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.

### 2.1.10 Debugging hook code

```
\DebugHooksOn
\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.
2.2 L3 programming layer (exp13) interfaces

This is a quick summary of the L\LaTeX3 programming interfaces for use with packages written in expl3. In contrast to the L\LaTeX2ε interfaces they always use mandatory arguments only, e.g., you always have to specify the \verb|⟨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.

\begin{verbatim}
\hook new:n \hook new_reversed:n \hook new_pair:nn
\hook disable:n \hook disable:n \hook disable:n \hook disable:n
\hook provide:n \hook provide:nn \hook provide_reversed:n
\hook provide_reversed:n \hook provide_reversed:n \hook provide_reversed:n
\hook use:n \hook use_once:n
\end{verbatim}

Creates a new \verb|⟨hook⟩| with normal or reverse ordering of code chunks. \verb|\hook new_pair:nn| creates a pair of such hooks with \verb|⟨⟨hook-2⟩⟩| being a reversed hook. If a hook name is already taken, an error is raised and the hook is not created.

The \verb|⟨hook⟩| can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

\begin{verbatim}
\hook disable:n \hook disable:n \hook disable:n \hook disable:n
\hook provide:n \hook provide:nn \hook provide_reversed:n
\hook provide_reversed:n \hook provide_reversed:n \hook provide_reversed:n
\hook use:n \hook use_once:n
\end{verbatim}

Like \verb|\hook new:n| but does nothing if the hook was previously declared with \verb|\hook new:n|. This declaration should only be used in special situations, e.g., when a command of another package needs to be altered and it is not clear if for that command a generic cmd hook was already explicitly declared before.

Normally \verb|\hook new:n| should be used instead.

\begin{verbatim}
\hook disable:n
\hook disable:n
\hook disable:n
\hook disable:n
\hook provide:n \hook provide:nn \hook provide_reversed:n
\hook provide_reversed:n \hook provide_reversed:n \hook provide_reversed:n
\hook use:n \hook use_once:n
\end{verbatim}

A shorthand for \verb|\hook provide:n(⟨hook-1⟩)\hook provide_reversed:n(⟨hook-2⟩)|.

\begin{verbatim}
\hook disable:n
\hook disable:n
\hook disable:n
\hook disable:n
\hook provide:n \hook provide:nn \hook provide_reversed:n
\hook provide_reversed:n \hook provide_reversed:n \hook provide_reversed:n
\hook use:n \hook use_once:n
\end{verbatim}

Executes the \verb|⟨⟨hook⟩⟩| code followed (if set up) by the code for next invocation only, then empties that next invocation code.

The \verb|⟨⟨hook⟩⟩| cannot be specified using the dot-syntax. A leading . is treated literally.

\begin{verbatim}
\hook disable:n
\hook disable:n
\hook disable:n
\hook disable:n
\hook provide:n \hook provide:nn \hook provide_reversed:n
\hook provide_reversed:n \hook provide_reversed:n \hook provide_reversed:n
\hook use:n \hook use_once:n
\end{verbatim}

Changes the \verb|⟨⟨hook⟩⟩| status so that from now on any addition to the hook code is executed immediately. Then execute any \verb|⟨⟨hook⟩⟩| code already set up.

The \verb|⟨⟨hook⟩⟩| cannot be specified using the dot-syntax. A leading . is treated literally.
\hook_gput_code:nnn \hook_gput_next_code:nn \hook_gremove_code:nn \hook_gset_rule:nnnn
\hook_if_empty:n TF

\hook_gput_code:nnn \{\hook}\{\label}\{\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 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 2.1.5.

\hook_gput_next_code:nn
\hook_gput_next_code:nn \{\hook\} \{\code\}
Adds a chunk of \code for use only in the next invocation of the \hook. 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 \code.

The \hook can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

\hook_gremove_code:nn
\hook_gremove_code:nn \{\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 2.1.5.

\hook_gset_rule:nnnn
\hook_gset_rule:nnnn \{\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 2.1.5. 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:n \\hook_if_empty:n TF
\hook_if_empty:n TF \{\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.
\hook_show:n \hook_show:n \{(hook)\}
\hook_log:n
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.

\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 (hook) can be specified using the dot-syntax to denote the current package name. See section 2.1.5.

\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}
  > 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, or example, you want to replace the code chunk for packageA, e.g.,

\RemoveFromHook{myhook}{packageA} \\
\AddToHook{myhook}{packageA}{\texttt{\textbackslash{}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 -> \texttt{\textbackslash{}typeout{A}}
  > packageB -> \texttt{\textbackslash{}typeout{B}}
  > packageC -> \texttt{\textbackslash{}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).

### 2.4 The use of “reversed” hooks

You may have wondered why you can declare a “reversed” hook with \texttt{\textbackslash{}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 \texttt{\textbackslash{}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 example\footnote{there are simpler ways to achieve the same effect}, suppose there is a package adding the following:

---

15
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{color}\end{itshape}

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}
  >  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.

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.

### 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:

16
• If \textit{(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 \textit{(code)} will never be executed.

• In contrast if that happens with a one-time hook the \textit{(code)} is executed immediately.

In particular this means that construct such as

\begin{verbatim}
\AddToHook{myhook}
  \{ \textit{(code-1)} \AddToHook{myhook}\{\textit{(code-2)}\} \textit{(code-3)} \}
\end{verbatim}

works for one-time hooks\footnote{This is sometimes used with \texttt{\AtBeginDocument} which is why it is supported.} (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 \texttt{\UseHook{myhook}} is executed it would

• execute \textit{(code-1)},

• then execute \texttt{\AddToHook{myhook}\{\textit{(code-2)}\}} which adds the code chunk \textit{(code-2)} to the hook for use on the next invocation,

• and finally execute \textit{(code-3)}.

The second time \texttt{\UseHook} is called it would execute the above and in addition \textit{(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 \textit{(code-2)} is added and so that code chunk is executed \(\#\ of\ invocations\) \(-\) 1 times.

\section{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 through 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: \texttt{\@kernel@before@\{hook\}} or \texttt{\@kernel@after@\{hook\}}. For example, in \texttt{\enddocument} you find

\begin{verbatim}
\UseHook{enddocument}\%
\@kernel@after@\enddocument
\end{verbatim}

which means first the user/package-accessible \texttt{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.\footnote{As 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!”}
2.7 Legacy \LaTeX\(2\varepsilon\) interfaces

\LaTeX\(2\varepsilon\) 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.

\AtBeginDocument\{\langle\text{label}\rangle\}\{\langle\text{code}\rangle\}

If used without the optional argument \langle label\rangle, it works essentially like before, i.e., it is adding \langle code\rangle to the hook \texttt{begindocument} (which is executed inside \begin{document}). However, all code added this way is labeled with the label \texttt{top-level} (see section 2.1.6) if done outside of a package or class or with the package/class name if called inside such a file (see section 2.1.5).

This way one can add further code to the hook using \texttt{\AddToHook} or \texttt{\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 \texttt{\AddToHook \{begindocument\} \langle\text{label}\rangle\{\langle\text{code}\rangle\}}.

\AtBeginDocument is a wrapper around the \texttt{begindocument} hook (see section 2.8.4), which is a one-time hook. As such, after the \texttt{begindocument} hook is executed at \begin{document} any attempt to add \langle code\rangle to this hook with \texttt{\AtBeginDocument} or with \texttt{\AddToHook} will cause that \langle code\rangle to execute immediately instead. See section 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\{\langle\text{label}\rangle\}\{\langle\text{code}\rangle\}

Like \texttt{\AtBeginDocument} but for the \texttt{enddocument} hook.

There is also \texttt{\AtBeginDvi} which is discussed in conjunction with the shipout hooks.

The few hooks that existed previously in \LaTeX\(2\varepsilon\) used internally commands such as \texttt{\@begindocumenthook} and packages sometimes augmented them directly rather than working through \texttt{\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 \texttt{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.

2.8 \LaTeX\(2\varepsilon\) commands and environments augmented by hooks

\textit{intro to be written}

2.8.1 Generic hooks for all environments

Every environment \langle\text{env}\rangle has now four associated hooks coming with it:

\texttt{env/\langle\text{env}\rangle/before} This hook is executed as part of \texttt{\begin} as the very first action, in particular prior to starting the environment group. Its scope is therefore not restricted by the environment.
\begin{itemize}
  \item **env/env\slash 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.
  \item **env/env\slash 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 third argument of \newenvironment).
  \item **env/env\slash 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/env\slash before and to env/env\slash after they can add surrounding environments and the order of closing them happens in the right sequence.

  Generic environment hooks are never one-time hooks even with environments that are supposed to appear only once in a document.\footnote{\textsuperscript{6}} In contrast to other hooks there is also no need to declare them 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 \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

  \begin{verbatim}
  \UseHook{env/quote/before}{quote \ldots \endquote}\UseHook{env/quote/after}
  \end{verbatim}

  to add the outer hooks, etc.
\end{itemize}

\texttt{\BeforeBeginEnvironment} \BeforeBeginEnvironment \((\langle label \rangle) \{\langle code \rangle\})
\begin{verbatim}
This declaration adds to the env/env\slash before hook using the \langle label \rangle. If \langle label \rangle is not given, the \langle default label \rangle is used (see section 2.1.5).
\end{verbatim}

\texttt{\AtBeginEnvironment} \AtBeginEnvironment \((\langle label \rangle) \{\langle code \rangle\})
\begin{verbatim}
Like \BeforeBeginEnvironment but adds to the env/env\slash begin hook.
\end{verbatim}

\texttt{\AtEndEnvironment} \AtEndEnvironment \((\langle label \rangle) \{\langle code \rangle\})
\begin{verbatim}
Like \BeforeBeginEnvironment but adds to the env/env\slash end hook.
\end{verbatim}

\texttt{\AfterEndEnvironment} \AfterEndEnvironment \((\langle label \rangle) \{\langle code \rangle\})
\begin{verbatim}
Like \BeforeBeginEnvironment but adds to the env/env\slash after hook.
\end{verbatim}

\footnote{Thus 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.}
2.8.2 Generic hooks for commands

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

\texttt{cmd/⟨name⟩/before} This hook is executed at the very start of the command execution.

\texttt{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 \texttt{after} hook works only with a subset of commands. Details about these restrictions are documented in \texttt{lcmdhooks-doc.pdf} or with code in \texttt{lcmdhooks-code.pdf}.

2.8.3 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 \texttt{\input}, \texttt{\include}, \texttt{\usepackage}, \texttt{\RequirePackage}, etc. These are documented in \texttt{ltfilehook-doc.pdf} or with code in \texttt{ltfilehook-code.pdf}.

2.8.4 Hooks provided by \texttt{\begin{document}}

Until 2020 \texttt{\begin{document}} offered exactly one hook that one could add to using \texttt{\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 \texttt{etoolbox} and others that augmented \texttt{\document} to gain better control.

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

\texttt{begindocument/before} This hook is executed at the very start of \texttt{\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 \texttt{etoolbox}’s \texttt{\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 2.5).

\texttt{begindocument} This hook is added to when using \texttt{\AtBeginDocument} and it is executed after the \texttt{.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.

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 2.5).

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

2.8.5 Hooks provided by \end{document}

\LaTeX{} 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 \LaTeX{} 2.09 it was not flexible enough for a number of use cases and so packages, such as etoolbox, averyend and others patched \end{document} 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 \end{document} 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):

\texttt{enddocument} The hook associated with \AtEndDocument. It is immediately called at the beginning of \end{document}.

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 2.5).

\texttt{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 2.5).

\texttt{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 2.5).

\texttt{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.

21
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 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 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, \LaTeX{} needs to be run several times, so initially it might get executed on the wrong page. See section 2.8.6 for where to find the details.

It is in 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 \LaTeX{} has finished the document processing.

### 2.8.6 Hooks provided by `\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`.

### 2.8.7 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., 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 have hooks in which such support can be added.

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

`sffamily` Like the `rmfamily` hook but for the `\sffamily` command.

`ttfamily` Like the `rmfamily` hook but for the `\ttfamily` command.

`normalfont` The `\normalfont` command resets 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 meta families (rm/sf/tt) and the meta series (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 meta families (rm/sf/tt) when \bfseries is called. In the bfseries/defaults hook further adjustments can 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 used in mdseries command.

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 |  
| \addtocmd | 3  
| \AddToHook | 1, 3, 5, 9, 13, 16, 17  
| \AddToHookNext | 4, 5, 9, 13  
| \AfterEndEnvironment | 19  
| \AtBeginDocument | 3, 7, 16–20  
| \AtBeginDvi | 18  
| \AtBeginEnvironment | 19  
| \AtEndDocument | 18, 20  
| \AtEndEnvironment | 19  
| \AtEndPreamble | 19  
| \BeforeBeginEnvironment | 19  
| \begin | 1, 18–20  
| \bfdefault | 22  
| \bfseries | 22  
| \ClearHookRule | 9  
| \clearpage | 20  
| \csname | 5  
| \DebugHooksOff | 11  
| \DebugHooksOn | 11  
| \DeclareDefaultHookRule | 9  
| \DeclareHookRule | 1, 4, 8, 9, 11, 13  
| \DeclareHookrule | 8  
| \DisableHook | 2  
| \document | 19, 20  
| \documentclass | 6, 7  
| \end | 1, 18, 20, 21  
| \endcsname | 17, 20  
| \enddocument | 10, 13  
| \errorstopmode |  
| \hook commands: |  
| \hook_debug_off: | 13  
| \hook_debug_on: | 13  
| \hook_disable:n | 11  
| \hook_gput_code:mmn | 12  
| \hook_gput_next_code:mmn | 12  
| \hook_gremove_code:mmn | 13  
| \hook_gset_rule:mmnn | 13  
| \hook_if_empty:nTF | 5, 6, 13  
| \hook_if_empty_p:n | 13  
| \hook_log:n | 13  
| \hook_new:n | 11, 12  
| \hook_new_pair:nn | 11  
| \hook_new_reversed:n | 11, 12  
| \hook_provide:n | 12  
| \hook_provide_pair:nn | 12  
| \hook_provide_reversed:n | 12  
| \hook_show:n | 13  
| \hook_use:n | 5, 6, 11, 12, 16  
| \hook_use_once:n | 5, 12, 16  
| \IfHookEmptyTF | 5, 6, 9  
| \ignorespaces | 20  
| \immediate | 20  
| \include | 19  
| \input | 7, 19  
| \listfiles | 21  
| \LogHook | 10  

23
<table>
<thead>
<tr>
<th>Command</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\SetDefaultHookLabel</td>
<td>5-7</td>
</tr>
<tr>
<td>\sffamily</td>
<td>21, 22</td>
</tr>
<tr>
<td>\showhook</td>
<td>11</td>
</tr>
<tr>
<td>\newenvironment</td>
<td>18</td>
</tr>
<tr>
<td>\small</td>
<td>4</td>
</tr>
<tr>
<td>\special</td>
<td>21</td>
</tr>
<tr>
<td>\newhook</td>
<td>2, 9, 15, 18</td>
</tr>
<tr>
<td>\newmirroredhookpair</td>
<td>2</td>
</tr>
<tr>
<td>\newreversedhook</td>
<td>2, 7, 15</td>
</tr>
<tr>
<td>\normalfont</td>
<td>22</td>
</tr>
<tr>
<td>\normalsize</td>
<td>4</td>
</tr>
<tr>
<td>\providehook</td>
<td>2</td>
</tr>
<tr>
<td>\providemirroredhookpair</td>
<td>4</td>
</tr>
<tr>
<td>\provided reversedhook</td>
<td>2</td>
</tr>
<tr>
<td>\pushdefaulthooklabel</td>
<td>6, 7</td>
</tr>
<tr>
<td>\begindocumenthook</td>
<td>18</td>
</tr>
<tr>
<td>\firstofone</td>
<td>3</td>
</tr>
<tr>
<td>\kernel@after@⟨hook⟩</td>
<td>17</td>
</tr>
<tr>
<td>\kernel@before@⟨hook⟩</td>
<td>17</td>
</tr>
<tr>
<td>\end</td>
<td>21</td>
</tr>
<tr>
<td>\expand@font@defaults</td>
<td>22</td>
</tr>
<tr>
<td>\ttfamily</td>
<td>22</td>
</tr>
<tr>
<td>\usetikzlibrary</td>
<td>7</td>
</tr>
<tr>
<td>\usepackage</td>
<td>6, 7, 19</td>
</tr>
<tr>
<td>\usetikzlibrary</td>
<td>7</td>
</tr>
<tr>
<td>\usetikzlibrary</td>
<td>7</td>
</tr>
<tr>
<td>\selectfont</td>
<td>22</td>
</tr>
<tr>
<td>\write</td>
<td>20</td>
</tr>
</tbody>
</table>