The \texttt{lthooks} package$^*$

Frank Mittelbach$^†$

August 31, 2020

Contents

1 Introduction 2

2 Package writer interface 2
  2.1 \LaTeX\2ε interfaces ........................................ 2
    2.1.1 Declaring hooks and using them in code .................. 2
    2.1.2 Updating code for hooks .................................. 4
    2.1.3 Hook names and default labels ............................. 5
    2.1.4 Defining relations between hook code .................... 6
    2.1.5 Querying hooks ............................................ 7
    2.1.6 Displaying hook code ...................................... 8
    2.1.7 Debugging hook code ...................................... 9
  2.2 L3 programming layer (\texttt{expl3}) interfaces ............... 9
  2.3 On the order of hook code execution .......................... 11
  2.4 The use of “reversed” hooks .................................. 12
  2.5 Difference between “normal” and “one-time” hooks ............. 13
  2.6 Private \LaTeX\ kernel hooks ................................... 14
  2.7 Legacy \LaTeX\2ε interfaces ................................... 14
  2.8 \LaTeX\2ε commands and environments augmented by hooks ....... 15
    2.8.1 Generic hooks for all environments ...................... 15
    2.8.2 Hooks provided by \texttt{\begin{document}} ................. 16
    2.8.3 Hooks provided by \texttt{\end{document}} ................... 17
    2.8.4 Hooks provided \texttt{\shipout} operations ................. 18
    2.8.5 Hooks provided by file loading operations ................. 18
    2.8.6 Hooks provided in NFSS commands ......................... 18

$^*$This package has version v1.0a dated 2020/08/31, © \LaTeX\ Project.

$^†$Code improvements for speed and other goodies by Phelype Oleinik
3 The Implementation
3.1 Loading further extensions .................................. 19
3.2 Debugging .................................................. 19
3.3 Borrowing from internals of other kernel modules .......... 20
3.4 Declarations ............................................... 20
3.5 Providing new hooks ....................................... 21
3.6 Parsing a label .............................................. 23
3.7 Setting rules for hooks code .............................. 31
3.8 Specifying code for next invocation ...................... 44
3.9 Using the hook .............................................. 44
3.10 Querying a hook .......................................... 46
3.11 Messages ................................................... 47
3.12 \LaTeX2ε package interface commands .................... 48
3.13 Internal commands needed elsewhere .................... 51

Index 52

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 \LaTeX2ε 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 \LaTeX{} 2ε 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 \texttt{\begin} and \texttt{\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.

\begin{Verbatim}
\texttt{\NewHook \langle hook \rangle}
\end{Verbatim}

Creates a new \langle hook \rangle. If this is a hook provided as part of a package it is suggested that the \langle hook \rangle name is always structured as follows: \langle package-name\rangle/\langle hook-name \rangle. 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 \langle hook \rangle can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

\begin{Verbatim}
\texttt{\NewReversedHook \langle hook \rangle}
\end{Verbatim}

Like \texttt{\NewHook} declares a new \langle hook \rangle, 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 \langle hook \rangle can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

\begin{Verbatim}
\texttt{\NewMirroredHookPair \langle hook-1 \rangle \langle hook-2 \rangle}
\end{Verbatim}

A shorthand for \texttt{\NewHook\langle hook-1\rangle}\texttt{\NewReversedHook\langle hook-2\rangle}.

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

\begin{Verbatim}
\texttt{\UseHook \langle hook \rangle}
\end{Verbatim}

Execute the hook code inside a command or environment.

Before \texttt{\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 \texttt{\begin\{document\}}. The \langle hook \rangle cannot be specified using the dot-syntax. A leading . is treated literally.

\begin{Verbatim}
\texttt{\UseOneTimeHook \langle hook \rangle}
\end{Verbatim}

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

\textit{FMi: Maybe add an error version as well?}

The \langle hook \rangle cannot be specified using the dot-syntax. A leading . is treated literally.
2.1.2 Updating code for hooks

\AddToHook \{\langle hook\rangle\}\{\langle label\rangle\}\{\langle code\rangle\}

Adds \langle code\rangle to the \langle hook\rangle labeled by \langle label\rangle. If the optional argument \langle label\rangle 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 \langle label\rangle then the new \langle code\rangle is appended to the existing one (even if this is a reversed hook). If you want to replace existing code under the \langle label\rangle, 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 \langle code\rangle 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 \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.

\RemoveFromHook \{\langle hook\rangle\}\{\langle label\rangle\}

Removes any code labeled by \langle label\rangle from the \langle hook\rangle. If the optional argument \langle label\rangle 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 \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.

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

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

\AddToHook{env/quote/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 \{\langle hook\rangle\}\{\langle code\rangle\}

Adds \langle code\rangle to the next invocation of the \langle hook\rangle. 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\{\langle hook\rangle\}\{\langle code-1\rangle\}\AddToHookNext\{\langle hook\rangle\}\{\langle code-2\rangle\}\}

will execute \langle code-1\rangle next time the \langle hook\rangle is used and at that point puts \langle code-2\rangle into the \langle hook\rangle 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 2.1.5.

The \langle hook\rangle 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 \textit{without specifying a} \langle label\rangle because then the package or class name is automatically used, which is helpful if rules are needed, and avoids mistyping the \langle label\rangle.

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 (.), or starts with a dot followed by a slash (./) then the dot denotes the \textit{default label} (usually the current package or class name—see \texttt{\DeclareDefaultHookLabel}). A "." or "/" anywhere else in a \langle hook\rangle or in \langle label\rangle is treated literally and is not replaced.

For example, inside the package \texttt{mypackage.sty}, the default label is \texttt{mypackage}, so the instructions:

\begin{verbatim}
\NewHook {./hook}
\AddToHook {./hook}\{.\}{code} % Same as \AddToHook{./hook}{code}
\AddToHook {./hook}\{./sub\}{code}
\DeclareHookRule{begindocument}{.}{before}{babel}
\AddToHook {file/after/foo.tex}{code}
\end{verbatim}

are equivalent to:

\begin{verbatim}
\NewHook {mypackage/hook}
\AddToHook {mypackage/hook}\{mypackage\}{code}
\end{verbatim}
The ⟨default label⟩ 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 ⟨default label⟩.

This syntax is available in all ⟨label⟩ arguments and most ⟨hook⟩, 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) and \IfHookExistsTF (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 ⟨label⟩, then the ⟨default label⟩ can be set using \DeclareDefaultHookLabel:

\DeclareDefaultHookLabel \DeclareDefaultHookLabel {(⟨default label⟩)}

Sets the ⟨default label⟩ to be used in ⟨label⟩ 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 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.
\DeclareHookRule {\langle hook\rangle}{\langle label1\rangle}{\langle relation\rangle}{\langle label2\rangle}

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

Currently, the supported relations are the following:

- before or < Code for \langle label1\rangle comes before code for \langle label2\rangle.
- after or > Code for \langle label1\rangle comes after code for \langle label2\rangle.

\begin{itemize}
  \item \textbf{incompatible-warning} Only code for either \langle label1\rangle or \langle label2\rangle can appear for that hook (a way to say that two packages—or parts of them—are incompatible). A warning is raised if both labels appear in the same hook.
  \item \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.
  \item \textbf{voids} Code for \langle label1\rangle overwrites code for \langle label2\rangle. More precisely, code for \langle label2\rangle is dropped for that hook. This can be used, for example if one package is a superset in functionality of another one and therefore wants to undo code in some hook and replace it with its own version.
  \item \textbf{unrelated} The order of code for \langle label1\rangle and \langle label2\rangle is irrelevant. This rule is there to undo an incorrect rule specified earlier.
\end{itemize}

There can only be a single relation between two labels for a given hook, i.e., a later \texttt{\DeclareHookRule} overwrites any previous declaration.

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.

\ClearHookRule \langle hook\rangle{\langle label1\rangle}{\langle label2\rangle}

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

\DeclareDefaultHookRule \langle label1\rangle{\langle relation\rangle}{\langle label2\rangle}

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

Declaring default rules is only supported in the document preamble.\footnote{Trying to do so, e.g., via \texttt{\DeclareHookRule} with ?? has bad side-effects and is not supported (though not explicitly caught for performance reasons).}

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

\subsection{2.1.5 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: 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 $\texttt{\textbackslash NewHook}$ or some variant thereof.

\begin{verbatim}
\IfHookEmptyTF \langle hook \rangle \texttt{(true code)} \texttt{(false code)}
\end{verbatim}

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

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

\begin{verbatim}
\IfHookExistsTF \langle hook \rangle \texttt{(true code)} \texttt{(false code)}
\end{verbatim}

Tests if the $\langle hook \rangle$ exists (if it was created with either $\texttt{\textbackslash NewHook}$, $\texttt{\textbackslash NewReversedHook}$, or $\texttt{\textbackslash NewMirroredHookPair}$), 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 $\langle hook \rangle$ 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.
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.

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

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 LATEX3 programming interfaces for use with packages written in expl3. In contrast to the LATEX2ε 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.

\hook_new:n \hook_new_reversed:n \hook_new_pair:nn

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

The (hook) can be specified using the dot-syntax to denote the current package name. See section 2.1.3.

\hook_use:n \hook_use_once:n

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

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

\hook_use_once:n

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

FMi: better L3 name?

The (hook) cannot be specified using the dot-syntax. A leading . is treated literally.
\hook_gput_code:nnn \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 2.1.3.

\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 2.1.3.

\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 2.1.3.

\hook_gset_rule:nnnn \hook_gset_rule:nnnn \{\hook\} \{\{\label1\}\} \{\{\relation\}\} \{\{\label2\}\}

Relate \{\label1\} with \{\label2\} when used in \{\hook\}. See \texttt{\DeclareHookRule} for the allowed \{\relation\}s. If \{\hook\} is \texttt{??} 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:n \hook_if_empty:n \{\hook\} \{\{true code\}\} \{\{false code\}\}

\hook_if_empty:n \hook_if_empty:n \{\hook\} \{\{true code\}\} \{\{false code\}\}

Tests if the \{\hook\} is empty (\textit{i.e.}, no code was added to it using either \texttt{\AddToHook} or \texttt{\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 \texttt{.} is treated literally.
\hook_if_exist:nTF \{\hook\} \{\{true code\}\} \{\{false code\}\}

Tests if the \(\langle\hook\rangle\) exists (if it was created with either \NewHook, \NewReversedHook, or \NewMirroredHookPair), and branches to either \(\langle true code\rangle\) or \(\langle false code\rangle\) 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 \(\langle\hook\rangle\) cannot be specified using the dot-syntax. A leading . is treated literally.

\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 \(\langle\hook\rangle\) under different labels are supposed to be independent if there are no special rules set up that define a relation between the chunks. This means that you can’t make assumptions about the order of execution!

Suppose you have the following declarations:

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

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

The hook 'myhook':
Code chunks:
  packageA -> $\typeout {A}$
  packageB -> $\typeout {B}$
  packageC -> $\typeout {C}$
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 example\(^3\), 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:

\(^3\)there are simpler ways to achieve the same effect.
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{color}\end{itshape}
```

and an error message that `\begin{color}{blue}` 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' :

<table>
<thead>
<tr>
<th>Code chunks:</th>
</tr>
</thead>
<tbody>
<tr>
<td>package-1 -&gt; \end {itshape}</td>
</tr>
<tr>
<td>top-level -&gt; \end {color}</td>
</tr>
</tbody>
</table>

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.

### 2.5 Difference between “normal” and “one-time” hooks

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

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

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

This has some consequences one needs to be aware of:

- If `(code)` is added to a normal hook after the hook was executed and it is never executed again for one or the other reason, then this new `(code)` will never be executed.

- In contrast if that happens with a one-time hook the `(code)` is executed immediately.

In particular this means that construct such as
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
\begin{itemize}
  \item execute \texttt{(code-1)},
  \item then execute \texttt{\AddToHook{myhook}{code-2}} which adds the code chunk \texttt{(code-2)} to the hook for use on the next invocation,
  \item and finally execute \texttt{(code-3)}.
\end{itemize}
The second time \texttt{\UseHook} is called it would execute the above and in addition \texttt{(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 \texttt{(code-2)} is added and so that code chunk is executed \texttt{(\# of invocations)} − 1 times.

\subsection{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@\langle hookname \rangle} or \texttt{\@kernel@after@\langle hookname \rangle}. 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!”}

\subsection{Legacy \LaTeX{}2ε interfaces}

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

With the new hook management several additional hooks have been added to \LaTeX{} and more will follow. See the next section for what is already available.
\AtBeginDocument \AtBeginDocument \{\langle label\rangle\} \{\langle 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 \texttt{\begin{document}}). However, all code added this way is labeled with the label \texttt{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 \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 the \texttt{top-level} code. When using the optional argument the call is equivalent to running \texttt{\AddToHook \{begindocument\} \{\langle label\rangle\} \{\langle code\rangle\}}.

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

\AtEndDocument \AtEndDocument \{\langle label\rangle\} \{\langle code\rangle\}

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

\AtBeginDvi \AtBeginDvi \{\langle label\rangle\} \{\langle code\rangle\}

This hook is discussed in conjunction with the shipout hooks.

The few hooks that existed previously in \LaTeX{} 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{} commands and environments augmented by hooks

\textit{intro to be written}

2.8.1 Generic hooks for all environments

Every environment \texttt{(env)} has now four associated hooks coming with it:

\texttt{env/(env)/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.

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

\texttt{env/(env)/end} This hook is executed as part of \texttt{\end} directly in front of the code specific to the end of the environment (e.g., the third argument of \texttt{\newenvironment}).
Generic environment hooks are never one-time hooks even with environments that are supposed to appear only once in a document. 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⟩ (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.

2.8.2 Hooks provided by \begin{document}

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

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

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

\begin{Verbatim}
\BeforeBeginEnvironment \BeforeBeginEnvironment [{(label)}] {⟨(code)⟩}
\AtBeginEnvironment \AtBeginEnvironment [{(label)}] {⟨(code)⟩}
\AtEndEnvironment \AtEndEnvironment [{(label)}] {⟨(code)⟩}
\AfterEndEnvironment \AfterEndEnvironment [{(label)}] {⟨(code)⟩}
\end{Verbatim}

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.

6Thus 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.
**begindocument** This hook is added to when using `\AtBeginDocument` and it 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.

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.

**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`.

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.

### 2.8.3 Hooks provided by `\end{document}`

\LaTeX{}2ε 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`, `afterend` and others patched `\enddocument` to add additional points where code could be hooked into.

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

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

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

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

**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.).

**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`.  

17
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 \texttt{\textbackslash listfiles} has been used and the warnings for duplicate labels, missing references, font substitutions etc.

Finally, this hook is executed just in front of the final call to \texttt{\textbackslash@@end}.

There is also the hook \texttt{shipout/lastpage}. This hook is executed as part of the last \texttt{\textbackslash shipout} in the document to allow package to add final \texttt{\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 \texttt{\textbackslash shipout}s is the last one, \LaTeX{} needs to be run several times, so initially it might get executed on the wrong page. See section 2.8.4 for where to find the details.

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

### 2.8.4 Hooks provided \texttt{\textbackslash shipout} operations

There are several hooks and mechanisms added to \LaTeX{}'s process of generating pages. These are documented in \texttt{ltshipout-doc.pdf} or with code in \texttt{ltshipout-code.pdf}.

### 2.8.5 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}, etc. These are documented in \texttt{ltfilehook-doc.pdf} or with code in \texttt{ltfilehook-code.pdf}.

### 2.8.6 Hooks provided in NFSS commands

In languages that need to support for more than one script in parallel (and thus several sets of fonts), e.g., Latin and Japanese fonts, NFSS font commands, such as \texttt{\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.

- **\texttt{rmfamily}** After \texttt{\rmfamily} has done its initial checks and prepared a any font series update this hook is executed and only afterwards \texttt{\selectfont}.
- **\texttt{sffamily}** Like the \texttt{rmfamily} hook but for the \texttt{\sffamily} command.
- **\texttt{ttfamily}** Like the \texttt{rmfamily} hook but for the \texttt{\ttfamily} command.
- **\texttt{normalfont}** The \texttt{\normalfont} command resets font encoding family series and shape to their document defaults. It then executes this hook and finally calls \texttt{\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/itn). 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.

3 The Implementation

3.1 Loading further extensions

At the moment the whole module rolls back in one go, but if we make any modifications in later releases this will then need splitting.

\_\_\_hook\_debug\_bool
 Holds the current debugging state.
\bool_new:N \_\_\_hook\_debug\_bool
(End definition for \_\_\_hook\_debug\_bool.)

\hook\_debug\_on:  \hook\_debug\_off:  \_\_\_hook\_debug\_n:  \_\_\_hook\_debug\_gset:

\cs_new\_eq:NN \_\_\_hook\_debug\_n \_\_\_hook\_debug\_n \use\_\_none:n
\cs_new\_protected:NPn \_\_\_hook\_debug\_on:
{\bool\_gset\_true:N \_\_\_hook\_debug\_bool
 \_\_\_hook\_debug\_gset:
}
\cs_new\_protected:NPn \_\_\_hook\_debug\_off:
{\bool\_gset\_false:N \_\_\_hook\_debug\_bool
 \_\_\_hook\_debug\_gset:
}
\cs_new\_protected:NPn \_\_\_hook\_debug\_gset:
{\cs\gset\_protected\_Npx \_\_\_hook\_debug\_n \#1
{\bool\_if:N \_\_\_hook\_debug\_bool \{\#1\} }
}
(End definition for \_\_\_hook\_debug\_on: and others. These functions are documented on page 11.)
3.3 Borrowing from internals of other kernel modules

\__hook_str_compare:nn Private copy of \__str_if_eq:nn
\cs_new_eq:NN \__hook_str_compare:nn \__str_if_eq:nn
(End definition for \__hook_str_compare:nn.)

3.4 Declarations

\l__hook_tmpa_bool Scratch boolean used throughout the package.
\bool_new:N \l__hook_tmpa_bool
(End definition for \l__hook_tmpa_bool.)

\l__hook_return_tl \l__hook_tmpa_tl \l__hook_tmpb_tl Scratch variables used throughout the package.
\tl_new:N \l__hook_return_tl
\tl_new:N \l__hook_tmpa_tl
\tl_new:N \l__hook_tmpb_tl
(End definition for \l__hook_return_tl, \l__hook_tmpa_tl, and \l__hook_tmpb_tl.)

\g__hook_all_seq In a few places we need a list of all hook names ever defined so we keep track if them in this sequence.
\seq_new:N \g__hook_all_seq
(End definition for \g__hook_all_seq.)

\g__hook_removal_list_prop A token list to hold delayed removals.
\tl_new:N \g__hook_removal_list_tl
(End definition for \g__hook_removal_list_prop.)

\l__hook_cur_hook_tl Stores the name of the hook currently being sorted.
\tl_new:N \l__hook_cur_hook_tl
(End definition for \l__hook_cur_hook_tl.)

\l__hook_work_prop A property list holding a copy of the \g__hook_{hook}_code_prop of the hook being sorted to work on, so that changes don’t act destructively on the hook data structure.
\prop_new:N \l__hook_work_prop
(End definition for \l__hook_work_prop.)

\g__hook_hook_curr_name_tl \g__hook_name_stack_seq Default label used for hook commands, and a stack to keep track of packages within packages.
\tl_new:N \g__hook_hook_curr_name_tl
\seq_new:N \g__hook_name_stack_seq
(End definition for \g__hook_hook_curr_name_tl and \g__hook_name_stack_seq.)

\__hook_tmp:w Temporary macro for generic usage.
\cs_new_eq:NN \__hook_tmp:w ?
(End definition for \__hook_tmp:w.)

\tl_gremove_once:Nx Some variants of expl3 functions.
3.5 Providing new hooks

Hooks have a \langle name \rangle and for each hook we have to provide a number of data structures. These are

\n
\textit{\g__hook\_\langle name \rangle\_code\_prop} A property list holding the code for the hook in separate chunks. The keys are by default the package names that add code to the hook, but it is possible for packages to define other keys.

\textit{\g__hook\_\langle name \rangle\_rule\_\langle label1 \rangle\_\langle label2 \rangle\_tl} A token list holding the relation between \langle label1 \rangle and \langle label2 \rangle in the \langle name \rangle. The \langle labels \rangle are lexically (reverse) sorted to ensure that two labels always point to the same token list. For global rules, the \langle name \rangle is ??.

\textit{\g__hook\_\langle name \rangle\_code\_tl} The code that is actually executed when the hook is called in the document is stored in this token list. It is constructed from the code chunks applying the information.

\textit{\g__hook\_\langle name \rangle\_reversed\_tl} Some hooks are “reversed”. This token list stores a - for such hook so that it can be identified. The - character is used because \langle reversed \rangle1 is +1 for normal hooks and -1 for reversed ones.

\textit{\g__hook\_\langle name \rangle\_code\_next\_tl} Finally there is extra code (normally empty) that is used on the next invocation of the hook (and then deleted). This can be used to define some special behavior for a single occasion from within the document. This token list is called code\_next rather than next\_code because otherwise a hook called \langle name \rangle\_next would have its code-token list named \g__hook\_\langle name \rangle\_\langle name \rangle\_code\_next\_tl, which would clash with the next code-token list of a hook called \langle name \rangle.

\textit{(End definition for \g__hook\_\langle name \rangle\_code\_prop, \g__hook\_\langle name \rangle\_code\_tl, and \g__hook\_\langle name \rangle\_code\_next\_tl.)}

\textit{\hook\_new\_n} The \hook\_new\_n declaration declare a new hook and expects the hook \langle name \rangle as its argument, e.g., \texttt{begindocument}.

\n
We check for one of the internal data structures and if it already exists we complain.
Otherwise we add the hook name to the list of all hooks and allocate the necessary data structures for the new hook.

\seq_gput_right:Nn \g__hook_all_seq {#1}

This is only used by the actual code of the current hook, so declare it normally:

\tl_new:c { \g__hook_#1_code_tl }

Now ensure that the base data structure for the hook exists:

\__hook_declare:n {#1}

The \g__hook_{hook}_labels_clist holds the sorted list of labels (once it got sorted). This is used only for debugging.

\clist_new:c { \g__hook_#1_labels_clist }

Some hooks should reverse the default order of code chunks. To signal this we have a token list which is empty for normal hooks and contains a - for reversed hooks.

\tl_new:c { \g__hook_#1_reversed_tl }

The above is all in L3 convention, but we also provide an interface to legacy \LaTeX hooks of the form \texttt{@...hook}, e.g., \texttt{@begindocumenthook}. There have been a few of them and they have been added to using \texttt{\g@addto@macro}. If there exists such a macro matching the name of the new hook, i.e., \texttt{@\texttt{\textless hook-name\texttt{\textgreater}hook}} and it is not empty then we add its contents as a code chunk under the label legacy.

Warning: this support will vanish in future releases!

\__hook_include_legacy_code_chunk:n {#1}

(End definition for \texttt{\__hook_declare:n}. This function is documented on page 9.)

\__hook_declare:n

This function declares the basic data structures for a hook without actually declaring the hook itself. This is needed to allow adding to undeclared hooks. Here it is unnecessary to check whether both variables exist, since both are declared at the same time (either both exist, or neither).

\cs_new_protected:Npn \__hook_declare:n #1

\hook_new_reversed:n

Declare a new hook. The default ordering of code chunks is reversed, signaled by setting the token list to a minus sign.

\cs_new_protected:Npn \hook_new_reversed:n #1

If the hook already exists the above will generate an error message, so the next line should be executed (but it is — too bad).

\tl_gset:cn { \g__hook_#1_reversed_tl } { - }
A shorthand for declaring a normal and a (matching) reversed hook in one go.
\cs_new_protected:Npn \hook_new_pair:nn #1 #2 {
  \hook_new:n {#1} \hook_new_reversed:n {#2}
}

The \LaTeX/X legacy concept for hooks uses with hooks the following naming scheme in the code: \@[\@...\hook].
If this macro is not empty we add it under the label \texttt{legacy} to the current hook and then empty it globally. This way packages or classes directly manipulating commands such as \texttt{\@begindocumenthook} still get their hook data added.

\textbf{Warning: this support will vanish in future releases!}
\cs_new_protected:Npn \__hook_include_legacy_code_chunk:n #1
  \tl_if_exist:cT { @#1hook } {
    \tl_if_empty:cF { @#1hook }
      { \exp_args:Nnnv \__hook_hook_gput_code_do:nnn {#1} { legacy } { @#1hook } }
    \tl_gclear:c { @#1hook }
  }
\debug_suspend:
If the macro doesn’t exist (which is the usual case) then nothing needs to be done.
Of course if the legacy hook exists but is empty, there is no need to add anything under \texttt{legacy} the legacy label.
\tl_if_novalue:nTF {#1}
  \__hook_currname_or_default:n {#2}
\debug_resume:

\subsection{Parsing a label}
This macro checks if a label was given (not \c_novalue_tl), and if so, tries to parse the label looking for a leading \texttt{.} to replace for \@currname. Otherwise \__hook_currname_or_default:n is used to pick \@currname or the fallback value.
\cs_new:Npn \__hook_parse_label_default:nn #1 #2
  \tl_if_novalue:nTF {#1}
    { \__hook_currname_or_default:n {#2} }
Start by checking if the label is empty, which raises an error, and uses the fallback value. If not, split the label at a ./, if any, and check if no tokens are before the ./, or if the only character is a .. If these requirements are fulfilled, the leading . is replaced with __hook_currname_or_default:n. Otherwise the label is returned unchanged.

\cs_new:Npn __hook_parse_dot_label:nn #1 #2
\begin{verbatim}
{ \tl_if_empty:nTF {#1}
  { \msg_expandable_error:nnn { hooks } { empty-label } {#2} 
    \tl_if_empty:nTF {\str_if_eq:nnTF {#1} { . } { #1 } 
      { \__hook_currname_or_default:n {#1} } 
      \__hook_parse_dot_label:nn {#2} {#1 ./ \s__hook_mark } 
    }
  }
  \__hook_parse_dot_label:nn #1 #2
{ \tl_if_empty:nTF {#2}
  { \__hook_parse_dot_label_aux:ww {#1} \s__hook_mark } 
  \__hook_parse_dot_label:nn {#2} 
{ \tl_if_empty:nTF {#3}
  \__hook_parse_dot_label_cleanup:ww #2 ./ \s__hook_mark } 
\end{verbatim}

(End definition for __hook_parse_label_default:nn.)

Uses \g__hook_hook_curr_name_tl if it is set, otherwise tries \@currname. If neither is set, uses the fallback value #1 (usually top-level).

\cs_new:Npn \__hook_currname_or_default:n #1
\begin{verbatim}
{ \tl_if_empty:nTF \g__hook_hook_curr_name_tl
  \tl_if_empty:nTF \@currname
    \{ \@currname \}
  \{ \g__hook_hook_curr_name_tl \}
\end{verbatim}

(End definition for \__hook_currname_or_default:n.)
With \hook_gput_code:nnn\{hook\}\{\langle label\rangle\}\{\langle code\rangle\} a chunk of \langle code\rangle is added to an existing \langle hook\rangle labeled with \langle label\rangle.

\cs_new_protected:Npn \hook_gput_code:nnn #1 #2
\begin{Verbatim}
\exp_args:Nxx \__hook_gput_code:nnn
\{ \__hook_parse_label_default:nn {#1} \{ top-level \} \}
\{ \__hook_parse_label_default:nn {#2} \{ top-level \} \}
\end{Verbatim}
\cs_new_protected:Npn \__hook_gput_code:nnn #1 #2 #3
\begin{Verbatim}
\begin{document}
First check if the current \langle hook\rangle/\langle label\rangle pair was marked for removal, in which case \__--\hook_unmark_removal:nn is used to remove that mark (once). This may happen when a package removes code from another package which was not yet loaded: the removal order is stored, and at this stage it is executed by not adding to the hook.
\__hook_if_marked_removal:nnTF {#1} {#2}
\{ \__hook_unmark_removal:nn {#1} {#2} \}
\end{document}
\end{Verbatim}
If no removal is queued, we are free to add. Start by checking if the hook exists.
\__hook_if_exist:nTF {#1}
\begin{Verbatim}
\begin{document}
If so we simply add (or append) the new code to the property list holding different chunks for the hook. At \begin{document}this is then sorted into a token list for fast execution.
\__hook_hook_gput_code_do:nnn {#1} {#2} {#3}
\end{document}
However, if there is an update within the document we need to alter this execution code which is done by \__--\hook_update_hook_code:n. In the preamble this does nothing.
\__hook_update_update_hook_code:n {#1}
\end{document}
\end{Verbatim}
\begin{Verbatim}
\begin{document}
If the hook does not exist, however, before giving up try to declare it as a generic hook, if its name matches one of the valid patterns.
\__hook_try_declaring_generic_hook:nnn {#1} {#2} {#3}
\end{document}
\end{Verbatim}
\cs_generate_variant:Nn \__hook_gput_code:nnn { nxv }
\begin{Verbatim}
\begin{document}
This macro will unconditionally add a chunk of code to the given hook.
\__hook_gput_code_do:nnn {#1} {#2} {#3}
\end{document}
\end{Verbatim}
\begin{Verbatim}
\begin{document}
However, first some debugging info if debugging is enabled:
\__hook_debug:n{\iow_term:x{****~ Add~ to--
\hook_if_exist:nF {#1} \{ undeclared- \}
hook- #1- \{#2\}
\on@line\space \leftarrow \tl_to_str:n(#3)\}
\end{document}
\end{Verbatim}
\end{Verbatim}
\begin{Verbatim}
\begin{document}
Then try to get the code chunk labeled \#2 from the hook. If there’s code already there, then append \#3 to that, otherwise just put \#3.
\prop_get:cnNTF { g__hook_#1_code_prop } \{#2\} \l__hook_return_tl
\prop_gput:cno { g__hook_#1_code_prop } \{#2\}
\prop_gput:cno \{ \l__hook_return_tl \#3 \}
\end{Verbatim}
\end{Verbatim}
\end{Verbatim}
Often it may happen that a package A defines a hook `foo`, but package B, that adds code to that hook, is loaded before A. In such case we need to add code to the hook before it's declared.

\begin{verbatim}
\cs_new_protected:Npn \__hook_gput_undeclared_hook:nnn #1 #2 #3
\__hook_declare:n {#1}
\__hook_gput_code_do:nnn {#1} {#2} {#3}
\end{verbatim}

(End definition for `\__hook_gput_undeclared_hook:nnn`.)

These entry-level macros just pass the arguments along to the common `\__hook_trydeclaring_generic_hook:nNNnn` with the right functions to execute when some action is to be taken.

The wrapper `\__hook_trydeclaring_generic_hook:nnn` then defers `\hook_gput_code:nnn` if the generic hook was declared, or to `\__hook_gput_undeclared_hook:nnn` otherwise (the hook was tested for existence before, so at this point if it isn’t generic, it doesn’t exist).

The wrapper `\__hook_trydeclaring_generic_next_hook:nn` for next-execution hooks does the same: it defers the code to `\hook_gput_next_code:nn` if the generic hook was declared, or to `\__hook_gput_next_do:nn` otherwise.

\begin{verbatim}
\cs_new_protected:Npn \__hook_trydeclaring_generic_hook:nnn #1 #2 #3
\__hook_trydeclaring_generic_hook:nNNnn {#1}
\hook_gput_code:nnn \__hook_gput_undeclared_hook:nnn
\end{verbatim}

(End definition for `\__hook_gput_undeclared_hook:nnn`.)

`\__hook_if_file_hook:wTF` If not then check it is one of a predefined set for generic names. We also split off the second component to see if we have to make a reversed hook. In either case the function returns ⟨true⟩ for a generic hook and ⟨false⟩ in other cases.

\begin{verbatim}
\cs_new_protected:Npn \__hook_trydeclaring_generic_hook:nnn #1 #2 #3
\__hook_if_file_hook:wTF #1 \__hook_gput_undeclared_hook:nnn
\__hook_gput_next_code:nn \__hook_gput_next_do:nn
\end{verbatim}
After having declared the hook we check the second component (for file hooks) or the third component for environment hooks) and if it is on the list of components for which we should have declared a reversed hook we alter the hook data structure accordingly.

\prop_if_in:NnTF \c__hook_generics_reversed_ii_prop {#2} { \tl_gset:cn { g__hook_#5_reversed_tl } { - } } { \prop_if_in:NnT \c__hook_generics_reversed_iii_prop {#3} { \tl_gset:cn { g__hook_#5_reversed_tl } { - } } }

Now that we know that the hook is declared we can add the code to it.

\prg_return_true:
\prg_return_false:

(End definition for \__hook_try_declaring_generic_hook:nnn and others.)

\__hook_if_file_hook_p:w \__hook_if_file_hook:wTF checks if the argument is a valid file-specific hook (not, for example, file/before, but file/before/foo.tex). If it is a file-specific hook, then it executes the ⟨true⟩ branch, otherwise ⟨false⟩.

A file-specific hook is file/(position)/(name). If any of these parts don’t exist, it is a general file hook or not a file hook at all, so the conditional evaluates to ⟨false⟩. Otherwise, it checks that the first part is file and that the ⟨position⟩ is in the \c__hook_generics_file_prop.

A property list is used here to avoid having to worry with catcodes, because expl3’s file name parsing turns all characters into catcode-12 tokens, which might differ from hand-input letters.
When a file-specific hook is found, before being declared it is lightly normalized by \__hook_file_hook_normalize:n. The current implementation just replaces two consecutive slashes (//) by a single one, to cope with simple cases where the user did something like \def\input@path{{./mypath/}}, in which case a hook would have to be \AddToHook(file/after//mypath//file.tex).

\cs_new:Npn \__hook_file_hook_normalize:n #1
\cs_new:Npn \__hook_strip_double_slash:n #1
\cs_new:Npn \__hook_strip_double_slash:w #1 // #2 \s__hook_mark
\tl_if_empty:nTF {#2}
{#1}
{ \__hook_strip_double_slash:w #1 / #2 \s__hook_mark }

\prop_const_from_keyval:Nn \c__hook_generics_prop {env=,file=,package=,class=,include=}

\prop_const_from_keyval:Nn \c__hook_generics_reversed_ii_prop {after=,end=}
\prop_const_from_keyval:Nn \c__hook_generics_reversed_iii_prop {after=}
\prop_const_from_keyval:Nn \c__hook_generics_file_prop {before=,after=}

Some of the generic hooks are supposed to use reverse ordering, these are the following (only the second or third sub-component is checked):

\prop_const_from_keyval:Nn \c__hook_generics_reversed_ii_prop {after=,end=} 
\prop_const_from_keyval:Nn \c__hook_generics_reversed_iii_prop {after=} 
\prop_const_from_keyval:Nn \c__hook_generics_file_prop {before=,after=}

With \hook_gremove_code:nn{(hook)}{(label)} any code for (hook) stored under (label) is removed.
First check that the hook code pool exists. \texttt{\_\_hook\_if\_exist:nTF} isn't used here because it should be possible to remove code from a hook before its defined (see section 2.1.5).

Then remove the chunk and run \texttt{\_\_hook\_update\_hook\_code:n} so that the execution token list reflects the change if we are after \texttt{\begin{document}}.

\begin{verbatim}
\prop_gclear:c { g\_\_hook\_#1\_code\_prop }
\end{verbatim}

Check if the label being removed exists in the code pool. If it does, just call \texttt{\_\_hook\-_gremove\_code\_do:nn} to do the removal, otherwise mark it to be removed.

Finally update the code, if the hook exists.

\begin{verbatim}
\prop_get:cnTF { g\_\_hook\_#1\_code\_prop } (#2) \_\_hook\_return\_tl
{ \_\_hook\_gremove\_code\_do:nn }
{ \_\_hook\_mark\_removal:nn }
{#1} {#2}
\end{verbatim}

If the code pool for this hook doesn’t exist it means that nothing tried to add to it before, so we just queue this removal order for later.

\begin{verbatim}
{ \_\_hook\_mark\_removal:nn }{#1} {#2} }
\end{verbatim}
Unmarks \langle label \rangle (#2) to be removed from \langle hook \rangle (#1). \tl_gremove_once:Nx is used rather than \tl_gremove_all:Nx so that two additions are needed to cancel two marked removals, rather than only one.

\cs_new_protected:Npn \__hook_unmark_removal:nn #1 #2
\tl_gremove_once:Nx \g__hook_removal_list_tl
\__hook_removal_tl:nn {#1} {#2}

(End definition for \__hook_unmark_removal:nn.)

\__hook_if_marked_removal:nnTF Checks if the \g__hook_removal_list_tl contains the current \langle label \rangle (#2) and \langle hook \rangle (#1).
\prop_new:cn {g__hook_??_code_prop}
\prop_new:cn {g__hook_??_code_tl}
\prop_new:cn {g__hook_??_reversed_tl}
\__hook_removal_tl:nn Builds a token list with #1 and #2 which can only be matched by #1 and #2. The & anchors a removal, so that #1 can’t be mistaken by #2 and vice versa, and the two $ delimit the two arguments
\__hook_removal_tl:nn {#1} {#2}

(End definition for \__hook_removal_tl:nn.)

Initially these variables simply used an empty “label” name (not two question marks). This was a bit unfortunate, because then l3doc complains about __ in the middle of a command name when trying to typeset the documentation. However using a “normal” name such as default has the disadvantage of that being not really distinguishable from a real hook name. I now have settled for ?? which needs some gymnastics to get it into the csname, but since this is used a lot things should be fast, so this is not done with c expansion in the code later on.

\g__hook_??_code_tl isn’t used, but it has to be defined to trick the code into thinking that ?? is actually a hook.
\prop_new:c {g__hook_??_code_prop}
\prop_new:c {g__hook_??_code_tl}

Default rules are always given in normal ordering (never in reversed ordering). If such a rule is applied to a reversed hook it behaves as if the rule is reversed (e.g., after becomes before) because those rules are applied first and then the order is reversed.
\__hook_removal_tl:nn

(End definition for \g__hook_??_code_prop, \g__hook_??_code_tl, and \g__hook_??_reversed_tl.)
3.7 Setting rules for hooks code

FMi: needs docu correction given new implementation

With \hook_gset_rule:nnnn{⟨hook⟩}{⟨label1⟩}{⟨relation⟩}{⟨label2⟩} a relation is defined between the two code labels for the given ⟨hook⟩. The special hook ?? stands for any hook describing a default rule.

First we ensure the basic data structure of the hook exists:
\__hook_declare:n {#1}

Then we clear any previous relationship between both labels.
\__hook_rule_gclear:nnn {#1} {#2} {#4}

Then we call the function to handle the given rule. Throw an error if the rule is invalid.
\debug_suspend:
\cs_if_exist_use:cTF { __hook_rule_#3_gset:nnn }
\{ {#1} {#2} {#4} \}
\__hook_update_hook_code:n {#1}
\{ \msg_error:nnnnn { hooks } { unknown-rule } {#1} {#2} {#3} {#4} \}
\debug_resume:

(End definition for \hook_gset_rule:nnnn and \__hook_gset_rule:nnnn. This function is documented on page 10.)

Then we add the new rule. We need to normalize the rules here to allow for faster processing later. Given a pair of labels $l_A$ and $l_B$, the rule $l_A > l_B$ is the same as $l_B < l_A$ only presented differently. But by normalizing the forms of the rule to a single representation, say, $l_B < l_A$, reduces the time spent looking for the rules later considerably.

Here we do that normalization by using \(pdf)strcmp to lexically sort labels $l_A$ and $l_B$ to a fixed order. This order is then enforced every time these two labels are used together.

Here we use \__hook_label_pair:nn to build a string $l_B|l_A$ with a fixed order, and use \__hook_label_ordered:nnTF to apply the correct rule to the pair of labels, depending if it was sorted or not.
\__hook_rule_<_gset:nnn
\__hook_rule_after_gset:nnn #1#2#3
{ \tl_gset:cn { g__hook_#1_rule_ \__hook_label_pair:nn {#3} {#2} _tl } \__hook_label_ordered:nnTF {#3} {#2} { < } { > } }
\cs_new_eq:cN { __hook_rule_>_gset:nnn } \__hook_rule_after_gset:nnn
(End definition for \__hook_rule_before_gset:nnn and others.)
\__hook_rule_voids_gset:nnn
This rule removes (clears, actually) the code from label #3 if label #2 is in the hook #1.
\cs_new_protected:Npn \__hook_rule_voids_gset:nnn #1#2#3
{ \tl_gset:cx { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _tl } \__hook_label_ordered:nnTF {#2} {#3} { -> } { <- } }
(End definition for \__hook_rule_voids_gset:nnn.)
\__hook_rule_incompatible-error_gset:nnn
\__hook_rule_incompatible-warning_gset:nnn
These relations make an error/warning if labels #2 and #3 appear together in hook #1.
\cs_new_protected:cpn { __hook_rule_incompatible-error_gset:nnn } #1#2#3
{ \tl_gset:cn { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _tl } { xE } }
\cs_new_protected:cpn { __hook_rule_incompatible-warning_gset:nnn } #1#2#3
{ \tl_gset:cn { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _tl } { xW } }
(End definition for \__hook_rule_incompatible-error_gset:nnn and \__hook_rule_incompatible-warning_gset:nnn.)
\__hook_rule_unrelated_gset:nnn
\__hook_rule_gclear:nnn
Undo a setting. \__hook_rule_unrelated_gset:nnn doesn’t need to do anything, since we use \__hook_rule_gclear:nnn before setting any rule.
\cs_new_protected:Npn \__hook_rule_unrelated_gset:nnn #1#2#3 { }
\cs_new_protected:Npn \__hook_rule_gclear:nnn #1#2#3
{ \cs_undefine:c { g__hook_#1_rule_ \__hook_label_pair:nn {#2} {#3} _tl } }
(End definition for \__hook_rule_unrelated_gset:nnn and \__hook_rule_gclear:nnn.)
\__hook_label_pair:nnn
Ensure that the lexically greater label comes first.
\cs_new:Npn \__hook_label_pair:nn #1#2
{ \if_case:w \__hook_str_compare:nn {#1} {#2} \exp_stop_f: #1 | #1 % 0 \or: #1 | #2 % 1 \else: #2 | #1 % -1 \fi: }
(End definition for \__hook_label_pair:nnn.)
\_\_hook_label_ordered_p:nn  Check that labels \#1 and \#2 are in the correct order (as returned by \_\_hook_label_-pair:nn) and if so return true, else return false.

\_\_hook_label_ordered:nnTF

\prg_new_conditional:Npnn \_\_hook_label_ordered:nn #1#2 { TF }
\{
  \if_int_compare:w \_\_hook_str_compare:nn {#1} {#2} > 0 \exp_stop_f:
    \prg_return_true:
  \else
    \prg_return_false:
  \fi:
\}

(End definition for \_\_hook_label_ordered:nnTF.)

\_\_hook_if_label_case:nnnnn  To avoid doing the string comparison twice in \_\_hook_initialize_single:NNNN (once with \str_if_eq:nn and again with \_\_hook_label_ordered:nn), we use a three-way branching macro that will compare \#1 and \#2 and expand to \use_i:nnn if they are equal, \use_ii:nn if \#1 is lexically greater, and \use_iii:nn otherwise.

\cs_new:Npn \_\_hook_if_label_case:nnnnn #1#2
\{\cs:w use_\if_case:w \_\_hook_str_compare:nn {#1} {#2} i \or: ii \else: iii \fi: :nnn\cs_end: \}

(End definition for \_\_hook_if_label_case:nnnnn.)

\_\_hook_update_hook_code:n  Before \begin{document} this does nothing, in the body it reinitializes the hook code using the altered data.

\cs_new_eq:NN \_\_hook_update_hook_code:n \use_none:n

(End definition for \_\_hook_update_hook_code:n.)

\_\_hook_initialize_all:  Initialize all known hooks (at \begin{document}), i.e., update the fast execution token lists to hold the necessary code in the right order.

\cs_new_protected:Npn \_\_hook_initialize_all: \{
  First we change \_\_hook_update_hook_code:n which so far was a no-op to now initialize one hook. This way any later updates to the hook will run that code and also update the execution token list.

  \cs_gset_eq:NN \_\_hook_update_hook_code:n \_\_hook_initialize_hook_code:n

  Now we loop over all hooks that have been defined and update each of them.

  \_\_hook_debug:n \{ \prop_gclear:N \g__hook_used_prop \}
  \seq_map_inline:Nn \g__hook_all_seq
  \{ \_\_hook_update_hook_code:n {##1} \}

  If we are debugging we show results hook by hook for all hooks that have data.

  \_\_hook_debug:n
  \{ \iow_term:x{\textasciitilde JAll- initialized- (non-empty)- hooks:} \_\_hook_used_prop
  \prop_map_inline:Nn \_\_hook_used_prop
  \{ \iow_term:x{\textasciitilde J- ##1- ->-} \exp_not:v \{\_\_hook_##1_code_tl}\} \}

\}
After all hooks are initialized we change the “use” to just call the hook code and not initialize it (as it was done in the preamble.

```latex
\cs_gset_eq:NN \hook_use:n \__hook_use_initialized:n
\cs_gset_eq:NN \__hook_preamble_hook:n \use_none:n
```

(End definition for \__hook_initialize_all:)

\__hook_initialize_hook_code:n

Initializing or reinitializing the fast execution hook code. In the preamble this is selectively done in case a hook gets used and at \begin{document} this is done for all hooks and afterwards only if the hook code changes.

```latex
\cs_new_protected:Npn \__hook_initialize_hook_code:n #1 {
\__hook_debug:n{ \iow_term:x{\texttt{Update code for hook \char\.'#1 \on@line}} }
```

This does the sorting and the updates. First thing we do is to check if a legacy hook macro exists and if so we add it to the hook under the label legacy. This might make the hook non-empty so we have to do this before the then following test.

```latex
\__hook_include_legacy_code_chunk:n {#1}
```

If there aren’t any code chunks for the current hook, there is no point in even starting the sorting routine so we make a quick test for that and in that case just update \texttt{g\_hook\_\langle hook\rangle\_code_tl} to hold the next code. If there are code chunks we call \texttt{\__hook\_initialize_single:NNNn} and pass to it ready made csnames as they are needed several times inside. This way we save a bit on processing time if we do that up front.

```latex
\hook_if_exist:nT {#1}
{
\prop_if_empty:cTF {g\__hook_#1_code_prop}
{ \tl_gset:co {g\__hook_#1_code_tl}
\cs:w g\__hook_#1_code_next_tl \cs_end: }
}{
```

By default the algorithm sorts the code chunks and then saves the result in a token list for fast execution by adding the code one after another using \texttt{\tl_gput_right:NV}. When we sort code for a reversed hook, all we have to do is to add the code chunks in the opposite order into the token list. So all we have to do in preparation is to change two definitions used later on.

```latex
\__hook_if_reversed:nTF {#1}
{ \cs_set_eq:NN \__hook_tl_gput:NV \tl_gput_left:NV
\cs_set_eq:NN \__hook_clist_gput:NV \clist_gput_left:NV }
{ \cs_set_eq:NN \__hook_tl_gput:NV \tl_gput_right:NV
\cs_set_eq:NN \__hook_clist_gput:NV \clist_gput_right:NV }
```

When sorting, some relations (namely voids) need to act destructively on the code property lists to remove code that shouldn’t appear in the sorted hook token list, so we temporarily save the old code property list so that it can be restored later.

```latex
\prop_set_eq:Nc \l__hook_work_prop { g\__hook_#1_code_prop }
\__hook_initialize_single:cccn
{ g\__hook_#1_code_tl } { g\__hook_#1_code_next_tl }
{ g\__hook_#1_labels_clist } {#1}
```

For debug display we want to keep track of those hooks that actually got code added to them, so we record that in plist. We use a plist to ensure that we record each hook name only once, i.e., we are only interested in storing the keys and the value is arbitrary.

34
\prop_new:N \g__hook_used_prop

(End definition for \__hook_initialize_hook_code:n.)

\g__hook_used_prop

All hooks that receive code (for use in debugging display).

(End definition for \g__hook_used_prop.)

\__hook_tl_csname:n \__hook_seq_csname:n

It is faster to pass a single token and expand it when necessary than to pass a bunch of character tokens around.

FMi: note to myself: verify

\seq_new:N \l__hook_labels_seq
\int_new:N \l__hook_labels_int
\tl_new:N \l__hook_front_tl
\tl_new:N \l__hook_rear_tl
\tl_new:c { \__hook_tl_csname:n { 0 } }

For the sorting I am basically implementing Knuth’s algorithm for topological sorting as given in TAOCP volume 1 pages 263–266. For this algorithm we need a number of local variables:

• List of labels used in the current hook to label code chunks:

\seq_new:N \l__hook_labels_seq

• Number of labels used in the current hook. In Knuth’s algorithm this is called \textit{N}:

\int_new:N \l__hook_labels_int

• The sorted code list to be build is managed using two pointers one to the front of the queue and one to the rear. We model this using token list pointers. Knuth calls them \textit{F} and \textit{R}:

\tl_new:N \l__hook_front_tl
\tl_new:N \l__hook_rear_tl

• The data for the start of the queue is kept in this token list, it corresponds to what Don calls \texttt{QLINK[0]} but since we aren’t manipulating individual words in memory it is slightly differently done:

\tl_new:c { \__hook_tl_csname:n { 0 } }

(End definition for \l__hook_labels_seq and others.)
\_hook\_initialize\_single:NNNn implements the sorting of the code chunks for a
hook and saves the result in the token list for fast execution (\#3). The arguments
are \langle hook-code-plist \rangle, \langle hook-code-tl \rangle, \langle hook-next-code-tl \rangle, \langle hook-ordered-labels-clist \rangle and
\langle hook-name \rangle (the latter is only used for debugging—the \langle hook-rule-plist \rangle is accessed using
the \langle hook-name \rangle).

The additional complexity compared to Don’s algorithm is that we do not use simple
positive integers but have arbitrary alphanumerics. As usual Don’s data structures
are chosen in a way that one can omit a lot of tests and I have mimicked that as far as
possible. The result is a restriction I do not test for at the moment: a label can’t be
equal to the number 0!

*FM: Needs checking for, just in case*

\cs_new_protected:Npn \_hook\_initialize\_single:NNNn #1#2#3#4 {
\debug_suspend:

**Step T1: Initialize the data structure …**

\seq_clear:N \l__hook_labels_seq
\int_zero:N \l__hook_labels_int

Store the name of the hook:

\tl_set:Nn \l__hook_cur_hook_tl {#4}

We loop over the property list holding the code and record all labels listed there.
Only rules for those labels are of interest to us. While we are at it we count them (which
gives us the \( N \) in Knuth’s algorithm). The prefix label_ is added to the variables to
ensure that labels named front, rear, labels, or return don’t interact with our code.

\prop_map_inline:Nn \l__hook_work_prop
{
\int_incr:N \l__hook_labels_int
\seq_put_right:Nn \l__hook_labels_seq {##1}
\tl_set:cn { \__hook_tl_csname:n {##1} }{0}
\seq_clear_new:c { \__hook_seq_csname:n {##1} }
}

**Steps T2 and T3: Sort the relevant rules into the data structure…**

This loop constitutes a square matrix of the labels in \_\_hook\_work\_prop in the
vertical and the horizontal directions. However since the rule \( l_A \langle rel \rangle l_B \)
is the same as \( l_B \langle rel \rangle l_A \) we can cut the loop short at the diagonal of the matrix (i.e., when both
labels are equal), saving a good amount of time. The way the rules were set up (see
the implementation of \_\_hook\_rule\_before\_gset:nnn above) ensures that we have no
rule in the ignored side of the matrix, and all rules are seen. The rules are applied
in \_\_hook\_apply\_label\_pair:nnn, which takes the properly-ordered pair of labels as
argument.

\prop_map_inline:Nn \l__hook_work_prop
{
\prop_map_inline:Nn \l__hook_work_prop
{
\__hook_if_label_case:nnnn {##1} {####1}
{ \prop_map_break: }
{ \__hook\_apply\_label\_pair:nnn {##1} {####1} }
{ \__hook\_apply\_label\_pair:nnn {####1} {##1} }
{#4}
}
}

36
Take a breath and take a look at the data structures that have been set up:

\_\_hook\_debug:n { \_\_hook\_debug\_label\_data:N \l__hook\_work\_prop }

Step T4:

\begin{verbatim}
\tl_set:Nn \l__hook\_rear\_tl { 0 }
\tl_set:cn { \_\_hook\_tl\_cname:n { 0 } } { 0 }
\seq_map_inline:Nn \l__hook\_labels\_seq
\begin{verbatim}
\int_compare:nNnT { \cs:w \_\_hook\_tl\_cname:n {##1} \cs_end: } = 0
\begin{verbatim}
\tl_set:cn { \_\_hook\_tl\_cname:n { \l__hook\_rear\_tl } }{##1}
\tl_set:Nn \l__hook\_rear\_tl {##1}
\end{verbatim}
\end{verbatim}
\end{verbatim}
\tl_set_eq:Nc \l__hook\_front\_tl { \_\_hook\_tl\_cname:n { 0 } }
\tl_gclear:N #1
\clist_gclear:N #3
\end{verbatim}

The whole loop combines steps T5–T7:

\begin{verbatim}
\bool_while_do:nn { ! \str_if_eq_p:Vn \l__hook\_front\_tl { 0 } }
\begin{verbatim}
\end{verbatim}
\end{verbatim}

This part is step T5:

\begin{verbatim}
\int_decr:N \l__hook\_labels\_int
\prop_get:NVN \l__hook\_work\_prop \l__hook\_front\_tl \l__hook\_return\_tl
\_\_hook\_tl\_gput:NV \l__hook\_return\_tl
\_\_hook\_clist\_gput:NV \l__hook\_front\_tl
\_\_hook\_debug:nf \iow_term:x{Handled- code- for- \l__hook\_front\_tl} }
\end{verbatim}

This is step T6 except that we don’t use a pointer \textit{P} to move through the successors, but instead use \texttt{##1} of the mapping function.

\begin{verbatim}
\seq_map_inline:cn { \_\_hook\_seq\_cname:n { \l__hook\_front\_tl } }
\begin{verbatim}
\int_compare:nNnT { \cs:w \_\_hook\_tl\_cname:n {##1} \cs_end: } = 0
\begin{verbatim}
\tl_set:cn { \_\_hook\_tl\_cname:n { \l__hook\_rear\_tl } }{##1}
\tl_set:Nn \l__hook\_rear\_tl {##1}
\end{verbatim}
\end{verbatim}
\end{verbatim}

and step T7:

\begin{verbatim}
\tl_set_eq:Nc \l__hook\_front\_tl
\begin{verbatim}
\end{verbatim}
\end{verbatim}

This is step T8: If we haven’t moved the code for all labels (i.e., if \l__hook\_labels\_int is still greater than zero) we have a loop and our partial order can’t be flattened out.

\begin{verbatim}
\int_compare:nNnF \l__hook\_labels\_int = 0
\begin{verbatim}
\end{verbatim}
\end{verbatim}

37
This is not really the information one needs in the error case but will do for now ...

FMi: fix

After we have added all hook code to \#1 we finish it off with adding extra code for a one time execution. That is stored in \#2 but is normally empty.

\tl_gput_right:Nn \#1 {\#2}
\debug_resume:
}
\cs_generate_variant:Nn \__hook_initialize_single:NNNn {ccc}
(End definition for \__hook_initialize_single:NNNn.)
\__hook_tl_gput:NV \__hook_clist_gput:NV
These append either on the right (normal hook) or on the left (reversed hook). This is setup up in \__hook_initialize_hook_code:n, elsewhere their behavior is undefined.
\cs_new:Npn \__hook_tl_gput:NV {\ERROR}
\cs_new:Npn \__hook_clist_gput:NV {\ERROR}
(End definition for \__hook_tl_gput:NV and \__hook_clist_gput:NV)
\__hook_apply_label_pair:nnn \__hook_if_exist_apply:nnnF
This is the payload of steps T2 and T3 executed in the loop described above. This macro assumes \#1 and \#2 are ordered, which means that any rule pertaining the pair \#1 and \#2 is \g__hook_⟨hook⟩_rule_\#1|\#2_tl, and not \g__hook_⟨hook⟩_rule_\#2|\#1_tl. This also saves a great deal of time since we only need to check the order of the labels once.

The arguments here are ⟨label1⟩, ⟨label2⟩, ⟨hook⟩, and ⟨hook-code-plist⟩. We are about to apply the next rule and enter it into the data structure. \__hook_apply_label_pair:nnn will just call \__hook_label_if_exist_apply:nnnF for the ⟨hook⟩, and if no rule is found, also try the ⟨hook⟩ name ?? denoting a default hook rule. \__hook_label_if_exist_apply:nnnF will check if the rule exists for the given hook, and if so call \__hook_apply_rule:nnn.
\cs_new_protected:Npn \__hook_apply_label_pair:nnn \#1\#2\#3
{ }
Extra complication: as we use default rules and local hook specific rules we first have to check if there is a local rule and if that exist use it. Otherwise check if there is a default rule and use that.
\__hook_label_if_exist_apply:nnnF \#1 \#2 \#3
{ }
If there is no hook-specific rule we check for a default one and use that if it exists.
\cs_new_protected:Npn \__hook_label_if_exist_apply:nnnF \#1 \#2 { ?? } { }
}
\ifcs_exist:w g__hook_ 3_rule_ #1 | #2 _tl \cs_end:
What to do precisely depends on the type of rule we have encountered. If it is a before rule it will be handled by the algorithm but other types need to be managed differently. All this is done in \__hook_apply_rule:nnn.

\__hook_apply_rule:nnn \{#1\} \{#2\} \{#3\}
\exp_after:wN \use_none:n
\else:
\use:nn
\fi:

(End definition for \__hook_apply_label_pair:nnn and \__hook_label_if_exist_apply:nnnF.)

\__hook_apply_rule:xE:nnn
\__hook_apply_rule:xW:nnn
These relations make two labels incompatible within a hook. xE makes raises an error if the labels are found in the same hook, and xW makes it a warning.

\__hook_apply_rule_xE:nnn
\__hook_apply_rule_xW:nnn
If we see \texttt{-} we have to drop code for label \#3 and carry on. We could do a little better and drop everything for that label since it doesn’t matter where we sort in the empty code. However that would complicate the algorithm a lot with little gain.\footnote{This also has the advantage that the result of the sorting doesn’t change which might otherwise (for unrelated chunks) if we aren’t careful.} So we still unnecessarily try to sort it in and depending on the rules that might result in a loop that is otherwise resolved. If that turns out to be a real issue, we can improve the code.

Here the code is removed from \texttt{l__hook_cur_hook_tl} rather than \#3 because the latter may be \texttt{??}, and the default hook doesn’t store any code. Removing from \texttt{l__hook_cur_hook_tl} makes default rules \texttt{-} and \texttt{<} work properly.

Reversed rules.

\section{Reversed rules.}

\texttt{\cs_new_eq:cc { \__hook_apply_-rule_<:nnn } { \__hook_apply_rule_>:nnn }} \hspace{1em} \texttt{\cs_new_eq:cc { \__hook_apply_-rule_>:nnn } { \__hook_apply_rule_<:nnn }} \hspace{1em} \texttt{\cs_new_eq:cc { \__hook_apply_-rule_<-:nnn } { \__hook_apply_rule_<-:nnn }} \hspace{1em} \texttt{\cs_new_eq:cc { \__hook_apply_-rule_->:nnn } { \__hook_apply_rule_->:nnn }} \hspace{1em} \texttt{\cs_new_eq:cc { \__hook_apply_-rule_xE:nnn } { \__hook_apply_rule_xE:nnn }} \hspace{1em} \texttt{\cs_new_eq:cc { \__hook_apply_-rule_xW:nnn } { \__hook_apply_rule_xW:nnn }}

\section{End definition for \texttt{\__hook_apply_-rule_<:nnn} and others.}
A macro to avoid moving this many tokens around.
\cs_new_protected:Npn \__hook_msg_pair_found:nnn #1 #2 #3
{
\iow_term:x{- \str_if_eq:nnTF {#3} {??} {default} {-normal} -
rule- \__hook_label_pair:nn {#1} {#2}:-
}\use:c { g__hook_#3_rule_ \__hook_label_pair:nn {#1} {#2} _tl } -
found}
(End definition for \__hook_msg_pair_found:nnn.)
\__hook_debug_label_data:N
\cs_new_protected:Npn \__hook_debug_label_data:N #1{
\iow_term:x{Code~ labels~ for~ sorting:}
\iow_term:x{- \seq_use:Nnnn \l__hook_labels_seq {~and~}{,~}{~and~} }
\prop_map_inline:Nn #1
{\iow_term:x{~ ##1~ =~ \tl_use:c{ \__hook_tl_csname:n {##1} }~ ->~
\seq_use:cnnn { \__hook_seq_csname:n {##1} }{~->~}{~->~}{~->} }
\iow_term:x{}}
(End definition for \__hook_debug_label_data:N.)
\hook_log:n
This writes out information about the hook given in its argument onto the terminal and
the .log file.
\cs_new_protected:Npn \hook_log:n #1
{
\exp_args:Nx \__hook_log:n
{ \__hook_parse_label_default:nn {#1} { top-level } }
}
\cs_new_protected:Npn \__hook_log:n #1
{
\__hook_preamble_hook:n {#1}
\iow_term:x{\textbf{The~ hook~ '#1':}}
\hook_if_exist:nF {#1}{\iow_term:x {~Hook~ is~ not~ declared!} }
\__hook_if_exist:nTF {#1}{\iow_term:x{-~Hook~ is~ not~ declared!} }
\__hook_if_exist:nTF {#1}{\iow_term:x{-~Hook~ is~ not~ declared!} }
\iow_term:x{-~Extra~ code~ next~ invocation:}
\iow_term:x{\@spaces
\tl_if_empty:cTF { g__hook_#1_code_next_tl }
\{---\}
\prop_if_empty:cTF {g__hook_#1_code_prop}
{\iow_term:x{\@spaces ---} }
{\prop_map_inline:cn {g__hook_#1_code_prop}
{\iow_term:x{\@spaces ##1~ \rightarrow~ \tl_to_str:n{##2} }}
\prop_map_inline:cn {g__hook_#1_code_prop}
{\iow_term:x{\@spaces ##1~ \rightarrow~ \tl_to_str:n{##2} }}
\prop_if_empty:cTF {g__hook_#1_code_next_tl}
{---}
If the token list is not empty we want to display it but without the first tokens (the code
to clear itself) so we call a helper command to get rid of them.

{-> \exp_args:Nv
 \__hook_log_next_code:n {g__hook_#1_code_next_tl} }

Loop through the rules in a hook and for every rule found, print it. If no rule is
there, print ---. The boolean \l__hook_tmpa_bool here indicates if the hook has no
rules.

\iow_term:x { -Rules: }
\bool_set_true:N \l__hook_tmpa_bool
\__hook_list_rules:nn {#1}
{ \bool_set_false:N \l__hook_tmpa_bool
 \iow_term:x
 { \@spaces ##2 with~
 \str_if_eq:nnT {##3} {??} { default~ }
 relation~ ##1 }
 }
\bool_if:NT \l__hook_tmpa_bool
{ \iow_term:x { @spaces --- } }
\bool_lazy_and:nnT
{ \hook_if_exist_p:n {#1} }
{ ! \hook_if_empty_p:n {#1} }
{ \iow_term:x { -Execution~ order
 \bool_if:NTF \l__hook_tmpa_bool
{ \__hook_if_reversed:nT {#1}
{ - (after~ reversal) }
 }
{ - (after~
 \__hook_if_reversed:nT {#1} {reversal~ and~}
 applying~ rules)
 }
}
\iow_term:x
{ \@spaces \clist_use:cn {g__hook_#1_labels_clist} { ,~ } }
}
{ \iow_term:n { -The~hook~is~empty. } }
\iow_term:n { }

To display the code for next invocation only (i.e., from \AddToHookNext we have to
remove the first two tokens at the front which are \tl_gclear:N and the token list to
clear.
\cs_new:Npn \__hook_log_next_code:n #1 {
 \exp_args:No \tl_to_str:n {\use_none:nn #1}
}

(End definition for \hook_log:n. This function is documented on page ??.)

This macro takes a ⟨hook⟩ and an ⟨inline function⟩ and loops through each pair of ⟨labels⟩
in the ⟨hook⟩, and if there is a relation between this pair of ⟨labels⟩, the ⟨inline function⟩
is executed with \#1 = \langle relation \rangle, \#2 = \langle label_1 \rangle/\langle label_2 \rangle, and \#3 = \langle hook \rangle (the latter may be the argument \#1 to \_\_\_hook_list_rules:nn, or ?? if it is a default rule).

\cs_new_protected:Npn \_\_\_hook_list_rules:nn #1 #2
\{
  \cs_set_protected:Npn \_\_\_hook_tmp:w #1 #2 #3 {#2}
  \prop_map_inline:cn { g__hook_#1_code_prop }
  { \prop_map_inline:cn { g__hook_#1_code_prop } }
  \_\_\_hook_if_label_case:nnnn #1 {#1} {#1} {#1}
  \prop_map_break: }
  \_\_\_hook_list_one_rule:nnn #1 {#1} {#1} {#1}
  \_\_\_hook_list_one_rule:nnn {#1} {#1} {#1}
  \_\_\_hook_list_rules:nn #1 #2
\}

These two are quite similar to \_\_\_hook_apply_label_pair:nnn and \_\_\_hook_\_\_label_if_exist_apply:nnnF, respectively, but rather than applying the rule, they pass it to the \langle inline function \rangle.

\cs_new_protected:Npn \_\_\_hook_list_one_rule:nnn #1#2#3
\{
  \_\_\_hook_list_if_rule_exists:nnnn #1 #2 #3 #1 \{ \}
  \_\_\_hook_list_if_rule_exists:nnnn #1 #2 #3 #1 \{ \}
\}

\cs_new_protected:Npn \_\_\_hook_list_if_rule_exists:nnnn #1#2#3#4
\{
  \if_cs_exist:w g__hook_#4_rule_#1 | #2 _tl \cs_end:
  \exp_args:Nv \_\_\_hook_tmp:w
  { g__hook_#4_rule_#1 | #2 _tl } { #1 | #2 } {#1}
\}

\_\_\_hook_debug_print_rules:n
A shorthand for debugging that prints similar to \prop_show:N.

\cs_new_protected:Npn \_\_\_hook_debug_print_rules:n #1
\{ \iow_term:n { The~hook~#1~contains~the~rules: } \cs_set_protected:Npn \_\_\_hook_tmp:w #1
\{
  \_\_\_hook_list_rules:nn #1 \{ \}
  \iow_term:x
  { \_\_\_hook_list_rules:nn #1 \{ \}
    \_\_\_hook_list_rules:nn #1 \{ \}
    \_\_\_hook_list_rules:nn #1 \{ \}
    \_\_\_hook_list_rules:nn #1 \{ \}
  }
  \_\_\_hook_list_rules:nn #1 \{ \}
  \exp_args:No \_\_\_hook_tmp:w \{ \use:nn \{ - \} \{ - \} \}
\}

(End definition for \_\_\_hook_debug_print_rules:n.)
3.8 Specifying code for next invocation

\hook_gput_next_code:nn

\cs_new_protected:Npn \hook_gput_next_code:nn #1
\exp_args:Nx \__hook_gput_next_code:nn { \__hook_parse_label_default:nn (#1) { top-level } }
\cs_new_protected:Npn \__hook_gput_next_code:nn #1 #2
\__hook_declare:n {#1}
\hook_if_exist:nTF {#1}
{ \__hook_gput_next_do:nn {#1} {#2} }
{ \__hook_try_declaring_generic_next_hook:nn {#1} {#2} }
\cs_new_protected:Npn \__hook_gput_next_do:nn #1 #2
\tl_if_empty:cT { g__hook_#2_code_next_tl }
{ \__hook_update_hook_code:n {#2} }
\tl_if_empty:NT #1
{ \tl_gset:Nn #1 { \tl_gclear:c { g__hook_#2_code_next_tl } } }
\tl_gput_right:Nn #1

First check if the “next code” token list is empty: if so we need to add a \tl_gclear:c to clear it, so the code lasts for one usage only. The token list is cleared early so that nested usages don’t get lost. \tl_gclear:c is used instead of \tl_gclear:N in case the hook is used in an expansion-only context, so the token list doesn’t expand before \tl_gclear:N: that would make an infinite loop. Also in case the main code token list is empty, the hook code has to be updated to add the next execution token list.

\cs_new_protected:Npn \__hook_gput_next_do:Nnn #1 #2
\tl_if_empty:cT { g__hook_#2_code_tl }
{ \__hook_update_hook_code:n {#2} }
\tl_if_empty:NT #1
{ \tl_gset:Nn #1 { \tl_gclear:c { g__hook_#2_code_next_tl } } }
\tl_gput_right:Nn #1

(End definition for \hook_gput_next_code:nn. This function is documented on page 10.)

3.9 Using the hook

\hook_use:n as defined here is used in the preamble, where hooks aren’t initialized by default. \__hook_use_initialized:n is also defined, which is the non-\protected version for use within the document. Their definition is identical, except for the \__hook_preamble_hook:n (which wouldn’t hurt in the expandable version, but it would be an unnecessary extra expansion).

\__hook_use_initialized:n holds the expandable definition while in the preamble. \__hook_preamble_hook:n initializes the hook in the preamble, and is redefined to \use_none:n at \begin{document}.

Both versions do the same internally: check if the hook exist as given, and if so use it as quickly as possible. If it doesn’t exist, the a call to \__hook_use:wn checks for file hooks.

At \begin{document}, all hooks are initialized, and any change in them causes an update, so \hook_use:n can be made expandable. This one is better not protected
so that it can expand into nothing if containing no code. Also important in case of
generic hooks that we do not generate a \relax as a side effect of checking for a csname.
In contrast to the \TeX low-level \csname ...\endcsname construct \tl_if_exist:c is
careful to avoid this.

\begin{verbatim}
\cs_new_protected:Npn \hook_use:n #1
\{
  \tl_if_exist:cTF { g__hook_#1_code_tl }
  { "\_hook_preamble_hook:n {#1}"
    \cs:w g__hook_#1_code_tl \cs_end:
  }
  { \_hook_use:wn #1 / \s__hook_mark {#1} }
}\}
\cs_new:Npn \_hook_use_initialized:n #1
\{
  \tl_if_exist:cTF { g__hook_#1_code_tl }
  { \cs:w g__hook_#1_code_tl \cs_end: }
  { \_hook_use:wn #1 / \s__hook_mark {#1} }
}\}
\cs_new_protected:Npn \_hook_preamble_hook:n #1
\{
\__hook_initialize_hook_code:n {#1} 
\}
\end{verbatim}

(End definition for \hook_use:n, \_hook_use_initialized:n, and \_hook_preamble_hook:n. This
function is documented on page 9.)

\_hook_use:wn does a quick check to test if the current hook is a file hook: those
need special treatment. If it is not, the hook does not exist. If it is, then \_hook-
try_file_hook:n is called, and checks that the current hook is a file-specific hook using
\_hook_if_file_hook:wTF. If it’s not, then it’s a generic file/ hook and is used if it exist.

If it is a file-specific hook, it passes through the same normalization as during decla-
ration, and then it is used if defined. \_hook_if_exist_use:n checks if the hook exist,
and calls \_hook_preamble_hook:n if so, then uses the hook.
\begin{verbatim}
\cs_new:Npn \_hook_use:wn #1 / #2 \s__hook_mark #3
\{
  \str_if_eq:nnTF {#1} { file }
  { \_hook_try_file_hook:n {#3} }
  { } % Hook doesn’t exist
}\}
\cs_new_protected:Npn \_hook_try_file_hook:n #1
\{
  \_hook_if_file_hook:wTF #1 / / \s__hook_mark
  { \exp_args:Ne \_hook_if_exist_use:n
    { \exp_args:Ne \_hook_file_hook_normalize:n {#1} }
    { \_hook_if_exist_use:n {#1} } % file/ generic hook (e.g. file/before)
  }
\}
\end{verbatim}
\hook_use_once:n

For hooks that can and should be used only once we have a special use command that remembers the hook name in \g__hook_execute_immediately_clist. This has the effect that any further code added to the hook is executed immediately rather than stored in the hook.

\begin{verbatim}
\cs_new_protected:Npn \hook_use_once:n #1
{ \tl_if_exist:cT \{ g__hook_#1_code_tl \}
  { \clist_gput_left:Nn \g__hook_execute_immediately_clist \{#1\}
    \hook_use:n \{#1\} }
}
\end{verbatim}

(End definition for \hook_use_once:n. This function is documented on page 9.)

\section{Querying a hook}

Simpler data types, like token lists, have three possible states; they can exist and be empty, exist and be non-empty, and they may not exist, in which case emptiness doesn't apply (though \tl_if_empty:N returns false in this case).

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 (even a hook that doesn’t exist may be non-empty).

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 (it may happen that a package \textit{A} defines a hook \texttt{foo}, but it’s loaded after package \textit{B}, which adds some code to that hook. In this case it is important that the code added by package \textit{B} is remembered until package \textit{A} is loaded).

A hook is said to exist when it was declared with \hook_new:n or some variant thereof.

\hook_if_empty:n, \hook_if_empty:TF

Test if a hook is empty (that is, no code was added to that hook). A hook being empty means that both its \g__hook_{\texttt{hook}}.code_prop and its \g__hook_{\texttt{hook}}.code_next_tl are empty.

\begin{verbatim}
\prg_new_conditional:Npn \hook_if_empty:n #1 { p , T , F , TF }
{ \__hook_if_exist:nTF \{#1\} }
\end{verbatim}
A canonical way to test if a hook exists. A hook exists if the token list that stores the
sorted code for that hook, \texttt{\g__hook\langle hook\rangle\_code\_tl}, exists. The property list \texttt{\g__hook\langle hook\rangle\_code\_prop} cannot be used here because often it is necessary to add code
to a hook without knowing if such hook was already declared, or even if it will ever be
(for example, in case the package that defines it isn’t loaded).

\begin{Verbatim}
\texttt{\prg_new_conditional:Nppnn \hook\_if\_exist:n \hook\_if\_exist:n\#1 { p , T , F , TF }
\{ \tl\_if\_exist:cTF \{ g\_\_hook\_\#1\_code\_tl \}
\{ \prg\_return\_true: \}
\{ \prg\_return\_false: \}
\}}
\end{Verbatim}

An internal check if the hook has already been declared with \_\_hook\_declare:n. This
means that the hook was already used somehow (a code chunk or rule was added to it),
but it still wasn’t declared with \hook\_new:n.

\begin{Verbatim}
\texttt{\prg_new_conditional:Nppnn \_\_\_hook\_if\_exist:n \_\_\_hook\_if\_exist:n\#1 { p , T , F , TF }
\{ \prop\_if\_exist:cTF \{ g\_\_hook\_\#1\_code\_prop \}
\{ \prg\_return\_true: \}
\{ \prg\_return\_false: \}
\}}
\end{Verbatim}

An internal conditional that checks if a hook is reversed.

\begin{Verbatim}
\texttt{\prg_new_conditional:Nppnn \_\_\_hook\_if\_reversed:p \_\_\_hook\_if\_reversed:n\#1 { p , T , F , TF }
\{ \if_int_compare:w \cs:w \g\_\_hook\_\#1\_reversed\_tl \cs_end: 1 < 0 \exp\_stop\_f:
\prg\_return\_true: \}
\else: \prg\_return\_false: \fi: \}}
\end{Verbatim}

List of hooks that from no on should not longer receive code.

\begin{Verbatim}
\texttt{\clist\_new:N \g\_\_hook\_execute\_immediately\_clist}
\end{Verbatim}

3.11 Messages

\begin{Verbatim}
\texttt{\msg\_new:nnnn \{ hooks \} \{ labels\-\_incompatible \}
\{ Labels\-‘\#1\’\-and\-‘\#2\’\-are\-incompatible
\str\_if\_eq:nnF \#3 \{ ?? \} \{ \texttt{-in\-hook\-‘\#3\’ } \} \-
\int\_compare:n\_nnTF \#4 = \{ 1 \}
\{ The\- code\- for both\- labels\- will\- be\- dropped. \}
\{ You\- may\- see\- errors\- later. \}}
\end{Verbatim}
\msg_new:nnnn { hooks } { exists } { Hook-‘#1’- has- already- been- declared. } { There- already- exists- a- hook- declaration- with- this- name. } Please- use- a- different- name- for- your- hook.

\msg_new:nnn { hooks } { empty-label } { Empty-code-label \msg_line_context:. Using-‘#1’-instead. }

\msg_new:nnnn { hooks } { unknown-rule } { Unknown- relationship- ‘#3’- between- labels- ‘#2’- and- ‘#4’- \str_if_eq:nF \{#1\} {??} { -in-hook-‘#1’ }. } Perhaps- a- misspelling?

\msg_new:nnn { hooks } { should-not-happen } {ERROR!-This-should-not-happen.-#1 \ }


3.12 \LaTeX\ 2\epsilon\ package interface commands

Declaring new hooks ...

\NewHook
\NewReversedHook
\NewMirroredHookPair

(End definition for \NewHook, \NewReversedHook, and \NewMirroredHookPair. These functions are documented on page 3.)

\AddToHook
\AddToHookNext

(End definition for \AddToHook. This function is documented on page 4.)
(End definition for \AddToHookNext. This function is documented on page 5.)

\RemoveFromHook
\NewDocumentCommand \RemoveFromHook { m o }
\{ \hook_gremove_code:nn {#1} {#2} \}

(End definition for \RemoveFromHook. This function is documented on page 4.)

\DeclareDefaultHookLabel
\__hook_curr_name_push:n
\__hook_curr_name_pop:
The token list \g__hook_hook_curr_name_tl stores the name of the current package/file to be used as label for hooks. Providing a consistent interface is tricky, because packages can be loaded within packages, and some packages may not use \DeclareDefaultHookLabel to change the default label (in which case \@currname is used, if set).

To pull that off, we keep a stack that contains the default label for each level of input. The bottom of the stack contains the default label for the top-level. Since the string top-level is hard-coded, here this item of the stack is empty. Also, since we’re in an input level, add lthooks to the stack as well. This stack should never go empty. An empty entry is added to the stack to account for the top-level. The item is empty so that the hard-coded default is used, but a call to \DeclareDefaultHookLabel will change it if needed.

\seq_gpush:Nn \g__hook_name_stack_seq { }

Two commands keep track of the stack: when a file is input, \__hook_curr_name_push:n pushes an (empty by default) label to the stack:
\cs_new_protected:Npn \__hook_curr_name_push:n #1
\seq_gpush:Nn \g__hook_name_stack_seq {#1}
\tl_gset:Nn \g__hook_hook_curr_name_tl {#1}

and when an input is over, the topmost item of the stack is popped, since the label will not be used again, and \g__hook_hook_curr_name_tl is updated to the now topmost item of the stack:
\seq_gpop:NN \g__hook_name_stack_seq \l__hook_return_tl
\tl_gget:NNTF \g__hook_name_stack_seq \l__hook_return_tl
\{ \tl_gset_eq:NN \g__hook_hook_curr_name_tl \l__hook_return_tl \}
\msg_error:nnn { hooks } { should-not-happen }
\{ Tried-to-pop-from-an-empty-default-label-stack. \}

The token list \g__hook_hook_curr_name_tl is but a mirror of the top of the stack. Now define a wrapper that replaces the top of the stack with the argument, and updates \g__hook_hook_curr_name_tl accordingly.
\NewDocumentCommand \DeclareDefaultHookLabel { m }
\{ \seq_gpop:NN \g__hook_name_stack_seq \l__hook_return_tl \__hook_curr_name_push:n {#1} \}
\EndAlignEnvironment{align*}
\begin{align*}  
  a &= b + c \\
  d &= e + f \\
  g &= h + i 
\end{align*}
\EndAlignEnvironment{align*}
\begin{equation}  
  a = b + c + \frac{d}{e} 
\end{equation}
\begin{equation}  
  a = b + c + \frac{d}{e} + \frac{f}{g} 
\end{equation}
\begin{equation}  
  a = b + c + \frac{d}{e} + \frac{f}{g} + \frac{h}{i} 
\end{equation}
\EndAlignEnvironment{align*}
3.13 Internal commands needed elsewhere

Here we set up a few horrible (but consistent) \TeX\ names to allow for internal commands to be used outside this module. We have to unset the @@ since we want double “at” sign in place of double underscores.

\ExplSyntaxOff
Rolling back here doesn’t undefine the interface commands as they may be used in packages without rollback functionality. So we just make them do nothing which may or may not work depending on the code usage.

If the hook management is not provided we make the test for existence false and the test for empty true in the hope that this is most of the time reasonable. If not a package would need to guard against running in an old kernel.
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

\[ \begin{array}{l}
\text{clist} & \text{new:N} & \text{........} & 633 \\
\text{clist} & \text{gclear:N} & \text{........} & 444 \\
\text{clist} & \text{gput_left:Nn} & \text{.......} & 388, 754 \\
\text{clist} & \text{gput_right:Nn} & \text{.......} & 390 \\
\text{clist} & \text{if_in:NnTF} & \text{.......} & 833 \\
\text{clist} & \text{new:N} & \text{........} & 48, 790 \\
\end{array} \]

\text{cs commands:}

\begin{array}{l}
\text{cs:w} & \text{........} & 353, 384, 436, 456, 459, 499, \\
& & 500, 501, 508, 515, 715, 722, 747, 784 \\
\text{cs_end:} & \text{........} & 356, 384, 436, 456, 459, 490, 500, \\
& & 501, 508, 515, 664, 715, 722, 747, 784 \\
\text{cs_process_variant:Nn} & \text{........} & 35, 88, 144, 478 \\
\text{cs_set:} & \text{........} & 360, 373, 374 \\
\text{cs_if_exist:use:N} & \text{........} & 302 \\
\text{cs_new:Nn} & \text{........} & 82, 89, 102, 112, \\
& & 113, 115, 225, 227, 229, 281, 335, \\
& & 351, 401, 402, 479, 480, 483, 505, 719, 727 \\
\text{cs_process_variant:Nn} & \text{........} & 7, 23, 34, 316, 322, 358, 554, \\
& & 555, 556, 557, 558, 880, 882, 884 \\
\text{cs_set:} & \text{........} & 7, 13, 18, 37, 42, 53, 61, \\
& & 65, 68, 125, 131, 145, 158, 163, 168, \\
& & 173, 182, 240, 246, 263, 265, 270, \\
& & 286, 297, 311, 317, 323, 328, 330, \\
& & 332, 333, 359, 376, 408, 481, 488, \\
& & 497, 504, 511, 518, 526, 532, 543, \\
& & 560, 567, 579, 584, 642, 657, 662, \\
& & 669, 685, 690, 697, 702, 710, 725, \\
& & 733, 742, 750, 842, 848, 865, 856, 867 \\
\text{cs_set:} & \text{........} & 387, 388, 389, 390 \\
\text{cs_set:} & \text{........} & 644, 672 \\
\text{cs_set:} & \text{........} & 334 \\
\end{array} \]

D

\text{debug commands:}

\begin{array}{l}
\text{debug_resume:} & \text{........} & 80, 309, 476 \\
\text{debug_suspend:} & \text{........} & 70, 301, 409 \\
\end{array} \]
\prop_if_empty_p:N \hfill 763
\prop_if_exist:NTF \hfill 778
\prop_if_in:NnTF \hfill 194, 197, 200, 218
\prop_map_break: \hfill 425, 650
\prop_map_inline:Nn \hfill 368, 413, 420, 422, 571, 596, 645, 647
\prop_new:N \hfill 31, 57, 283, 284, 400
\prop_put:Nnn \hfill 541, 552
\prop_set_eq:N \hfill 391
\prop_show:N \hfill 42
\providecommand \hfill 899, 902

Q
quark internal commands:
\s_hook_mark \hfill 36, 99, 102, 105, 109, 112, 113, 175, 209, 228, 229, 233, 717, 723, 727, 735
\relax \hfill 44
\RemoveFromHook \hfill 3, 3, 839, 901
\rmfamily \hfill 17

S
scan commands:
\scan_new:N \hfill 36
\scan_stop: \hfill 184, 189
\selectfont \hfill 17, 18
seq commands:
\seq_clear:N \hfill 410
\seq_clear_new:N \hfill 418
\seq_get:NNTF \hfill 851
\seq_gpop:NN \hfill 850, 860
\seq_gpush:Nn \hfill 841, 844
\seq_gput_right:Nn \hfill 45
\seq_map_inline:Nn \hfill 302, 434, 452
\seq_new:N \hfill 28, 33, 403
\seq_put_right:Nn \hfill 416, 509, 516
\seq_use:Nnnn \hfill 569, 574
\sfamily \hfill 17
\shipout \hfill 17
\ShowHook \hfill 8, 10, 12, 865, 906
\small \hfill 3
\space \hfill 150
\special \hfill 17
str commands:
\str_if_eq:nn \hfill 32
\str_if_eq:nnTF \hfill 97, 211, 250, 562, 613, 679, 729, 794, 811
\str_if_eq_p:nn \hfill 215, 445
str internal commands:
\__str_if_eq:nn \hfill 19, 23

T
\TeX\ and \LaTeX\ 2\epsilon \ commands:
\@hook \hfill 21, 22
\@begindocumenthook \hfill 14, 21, 22
\@currname \hfill 5, 5, 22, 23, 48, 119, 121
\@empty \hfill 907, 908
\@enddocumenthook \hfill 22
\@exp@hook@curr@name@pop@ \hfill 880
\@exp@hook@curr@name@push@n \hfill 880
\@exp@initialize@all@ \hfill 880
\@firstofone \hfill 2
\@gobble@AddToHook@args \hfill 898, 899
\@gobble@RemoveFromHook@args \hfill 901, 902
\@kernel@after@hookname \hfill 13
\@kernel@before@hookname \hfill 13
\@onlypreamble \hfill 872
\@spaces \hfill 594, 597, 600, 612, 617, 633
\@end \hfill 17
\@expand@font@defaults \hfill 18
\@addTo@macro \hfill 21
\@on@line \hfill 150, 378
\@protected \hfill 49
\tl commands:
\\c@novalue_tl \hfill 22
\\tl_gclear:N \hfill 41, 43, 77, 443, 707
\\tl_gput_left:Nn \hfill 387
\\tl_gput_right:Nn \hfill 33, 267, 359, 473, 708
\\tl_gremove_all:Nn \hfill 29
\\tl_gremove_once:Nn \hfill 29, 29, 35, 35, 272
\\tl_gset:Nn \hfill 63, 198, 201, 313, 319, 325, 329, 331, 383, 707, 845
\\tl_gset_eq:N \hfill 852
\\tl_if_empty:N \hfill 45
\\tl_if_empty:NNTF \hfill 73, 117, 119, 601, 704, 706
\\tl_if_empty:NTF \hfill 91, 104, 107, 191, 231
\\tl_if_empty_p:N \hfill 764
\\tl_if_empty_p:Nn \hfill 214
\\tl_if_exist:N \hfill 44
\\tl_if_exist:NNTF \hfill 71, 712, 721, 744, 752, 772
\\tl_if_in:NnTF \hfill 277
\\tl_if_novalue:NTF \hfill 84
\\tl_new:N \hfill 25, 26, 27, 29, 30, 32, 46, 49, 58, 285, 405, 406, 407
\\tl_set:Nn \hfill 412, 417, 432, 433, 468, 439, 454, 461, 462, 507, 514
\\tl_set_eq:N \hfill 442, 465
\\tl_to_str:n \hfill 150, 282, 597, 640
\\tl_trim_spaces_apply:N \hfill 86
\\tl_use:N \hfill 573
\ttfamily \hfill 17

56
<table>
<thead>
<tr>
<th>Command</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>\use_i:nn</td>
<td>32</td>
</tr>
<tr>
<td>\use_ii:nn</td>
<td>32</td>
</tr>
<tr>
<td>\use_iii:nn</td>
<td>32</td>
</tr>
<tr>
<td>\use:N</td>
<td>523, 524, 564</td>
</tr>
<tr>
<td>\use:n</td>
<td>288</td>
</tr>
<tr>
<td>\use:nn</td>
<td>494, 683</td>
</tr>
<tr>
<td>\use:nnn</td>
<td>32</td>
</tr>
<tr>
<td>\usepackage</td>
<td>5, 17</td>
</tr>
<tr>
<td>\UseHook</td>
<td>2, 5, 10, 12, 13, 863, 904</td>
</tr>
<tr>
<td>\UseOneTimeHook</td>
<td>2, 12, 863, 905</td>
</tr>
<tr>
<td>\use_none:n</td>
<td>7, 43, 358, 374, 492</td>
</tr>
<tr>
<td>\use_none:nn</td>
<td>640</td>
</tr>
</tbody>
</table>