Thoughts on Hooks in code

Frank Mittelbach LaTeX Project Team

2019-10-15

Contents

1 Introduction **1**
1.1 Hooks and their interface 1
1.2 Configuration points and their interface 3
1.3 Distinction between configuration points and hooks 3
1.4 Thoughts on current `l3hooks` interfaces 3
1.4.1 Next viz all (or what is called "repeated hooks") 3
1.4.2 Document viz repeated hooks 3
1.5 Thoughts on the `\end` (environment) interface 4

2 Hooks and config points in various places **5**
2.1 Document structure 5
2.2 Output routine 6
2.2.1 Making pages 6
2.2.2 Making columns 7
2.3 Heading commands 8

1 Introduction

This document collects some thoughts on places for hooks (i.e., code interfaces where other code can add material for execution).

I distinguish this from what I call "configuration points", which I think of as being commands that have a single definition that can be replaced by a different one but not "added to" by different packages. For example, "add begin document" is a typical hook, as it is a place where different packages my want to execute some code. On the other hand "prepare footins" is a configuration interface whose sole purpose is to do something specific to the `\footins` box. This may differ based on the intended outcome, but there can only one definition active at any time.[1]

[1] Technically this is of course not quite true, there is a certain gray area, but only if some packages very closely interrelated. The main point is that "hooks" are more about places where code gets added without (much) concern about what other package want to execute.

1.1 Hooks and their interface

At the moment most existing hooks in LaTeX 2_ε are actually named `\@...hook` (with one or two exception). We should keep that approach for backward compatibility and only change the exceptions.

Currently we do have declarations such as `\AtBeginDocument` to add data to a hook. The order in which the code gets executed then only depends on the order of the declarations, i.e., on the order of package loading.

What is not possible right now is define known dependencies between different code executed in a certain place, result in the famous "try changing the order of package X and Y to resolve the problem". Sometimes this works, but sometime a different order would be needed for different hooks and then of course that advice doesn't really any longer work.

As an alternative/extension we envision that material added to a hook will be named (using the package name by default) and that it will be possible for packages to declare rules that its code for a certain hook should be placed in relation to code from some other package (if that package is loaded). This way known dependencies could be automatically resolved (if declared as a rule by either side—or even both if they don't conflict) and impossible requirements could be detected as being incompatible.

Outline of interface:

```
\AddToHook{<hook>}{<name>}{<code>}
\ClearHook{<hook>}{<name>}              % maybe
\SetupHook{<hook>}{<name>}{<code>}      % \ClearHook followed by \AddToHook

\SetupHookRule{<hook>}{<name1>}{<name2>}{<rule>}
```

`\AddToHook` would add additional *code* labeled by *name* to the *hook*. If there exists already code under that *name* then it is appended. `\ClearHook` removes all code labeled by *name*. `\SetupHookRule` defines some relationship between *name1* and *name2*, for the exact syntax a lot different ways are possible (should be determined later).

One could also think of *name* to be optional defaulting to the current package/class name to encourage people to use that as the name normally.

[2019/06/30] One important aspect not covered above is the question of what is stored with respect to *code*. In most cases it should be the unexpanded code, but we will also need a version of all commands above that stored the expansion (protected expansion) in case values current at the time of setup need to be stored.

Existing hook interface commands such as `\AtBeginDocument` would still be supported as follows:

- They add their code under the name "`legacy`" to the hook.
- Other packages could then decide that they should come before or after the code labeled `legacy`.

In other words:

```
\def\AtBeginDocument#1{\AddToHook{begindocument}{legacy}{#1}}
\def\AtEnddocument#1{\AddToHook{enddocument}{legacy}{#1}}
```

However, there use would be discouraged in favor of explicitly labeling the hook code with the package name.

There also exist `\AtEndOfPackage` and `\AtEndOfClass` which uses

```
\@currname.\@currext-h@@k
```

as the hook name, but I don't think this needs to be incorporated into the more general mechanism as I don't think this being usable anywhere outside the actual package code—that one package writes into that hook of another package is in theory be possible (as in overwrite some code of that package if it gets loaded later than me, but I doubt that this makes a useful/explainable interface). On the other hand it is surprisingly often used (in 379 package/classes in TL 2019) so perhaps that assumption needs some checking.

1.2 Configuration points and their interface

As configuration points allow for only a single definition at any one time they can be simply defined as commands without providing an explicit interface.

My current suggestion is to all call them `\@...config` in the LaTeX 2_ε code and provide them with a default definition. We could think of offering a configuration interface such as

```
\SetupConfigPoint{@makecol}{preparefootins}{<code>}
```

but that wouldn't do much other than (globally) replacing the config command definition with the new one.

I also think that such config point commands should not take arguments even if they technically have inputs (like the current page box or the `\footins` box etc.).

1.3 Distinction between configuration points and hooks

As a rule of thumb, any place where the result of executing *code* requires a certain specific outcome then it is normally a configuration point and not a general hook, e.g., "the code write something into a certain box" or "the code runs a specific set of other code fragments and all it is supposed to do is to decide the order or not run some of them". There could be excepts and boundary cases but I think as a basic rule this makes sense.

In contrast if we don't make any (or not much) assumptions on what could be executed and what the side effects are supposed to be then it is most likely a general hook.

1.4 Thoughts on current l3hooks interfaces

In l3hooks we currently have the following type of interfaces that would need some merging with the thoughts above.

1.4.1 Next viz all (or what is called "repeated hooks")

First of all there is the idea of executing some code only once a hook is reached the next time, e.g., "do something on the next page" in contrast to "do something on every page". The question to resolve here seems to me what is the relationship between code of the different types. Probably the easiest way to handle this is to execute "next" code always after "all" code.

However, is this general enough or is more control needed?

1.4.2 Document viz repeated hooks

In short I'm not yet convinced that the distinction is useful, is it? If so how/why? Example please.

[JAW 2019-07-01] *The starting point for the split is that it makes sense to namespace document-wide hooks in one place: it does not really matter which module adds them. That of course could be handled by*

```
\hook_new:nn { document } { name-of-hook }
```

The second thing is that the next/all mechanism is redundant for such hooks. So I thought I'd just make a dedicated interface that did not have quite the same overhead.

The third and perhaps most important thing is that document hooks can only be used once and this affects adding to them. If you look at current `\AtBeginDocument`*, it can still be 'added to' after* `\document`*, but is a simple* `\@firstofone`*. To do that, the hook needs to 'know' it's a one-shot. I was aiming for something similar, although I've currently gone for an error rather than silently dumping the tokens. My thinking is that*

```
\hook_use:nn { document } { name }
\hook_gpush:nnnn { document } { name } { pkg } { content }
```

is going to be quite surprising.

1.5 Thoughts on the \end (environment) interface

From a mail by Will:

```
1.  It's interesting that breqn redefines \end or now ``\end `` to be:

      \@namedef{end }#1{\csname end#1\endcsname \latex@end{#1}}%

This allows an environment end definition like this:
```

```
      \def\enddmath#1{\check@punct@or@qed}

which ‘‘absorbs’’ the \latex@end (the #1), switches over to the aux
 function, which is

      \def\check@punct@or@qed#1{...
         \def\finish@end{\csname end@#1\endcsname\latex@end{#1}}%
         \check@punct
      }

which allows \futurelet lookahead and all that, using \finish@end
 finally to tidy up the environment properly.

This is all rather convoluted and very neat and tidy.

As far as I know there are no incompatibilities with breqn and any
 other major package, so the question arises whether it would be worth
 setting up "\end ‘‘ a little more directly to allow this behaviour. Of
 course it will slow down every environment in a document by one
 expansion, but is that a big deal? It may well be, I don't know.


2.  If we build a hook into 2e to provide this, do we add an interface
 to xparse to allow environments to lookahead? As well as being useful
 for breqn - and it's such a good part of breqn I'd argue it's worth
 breaking it out from there for xmath - wouldn't this style of
 lookahead be needed for LDB concepts?
```

2 Hooks and config points in various places

In this section we collect existing LaTeX 2_ε hooks (both by the kernel and/or by package) as well a hooks that do not exist but would be beneficial to have for one or the other reason. As of now it mainly discusses the LaTeX 2_ε situation.

For each hook/config point we document

- a “name”;
- the area where it applies;
- whether it is “new” or does already exist (if so where);
- and the major reason why it would be beneficial.

If there are several distinct use cases each could be added using `\item`. You can cross reference hooks using `\label`/`\ref`.

2.1 Document structure

Hook (1): documentclass (doc-structure/*kernel*)

- This is really more or less internal (used for handling 2.09 compatibility) but it shows up in one or two other places in TL.

Hook (2): begindocument (`\document`/*kernel*)

-

Hook (3): enddocument (`\enddocument`/*kernel*)

-

2.2 Output routine

2.2.1 Making pages

Hook (4): begindvibox (`\@outputpage`/*kernel*)

- `\AtBeginDvi` is a legacy LaTeX 2_ε interface that hooks into the first shipout box at the very top. Afterwards the material is dropped so it doesn't show up in later boxes. What is special is that the hook itself is implemented as a box and each time `\AtBeginDvi` is called more material is added to that box which eventually is unboxed into the first shipout box. in that respect that hook is rather "special" and right now I'm not sure it really has to—use cases please.

- If not then my proposal would be to deprecate it in favor of a "normal" hook at the same place (i.e., implemented as a token list) that would fit into the general model outlined above.

2019/06/30 The current thinking is that the use of a box for storage was mainly done to help preserving space in the early LaTeX 2_ε days, so it should be possible to drop that in favor of a hook where the code is stored like any other hook in form of a token list.

Hook (5): firstshipout (`\@outputpage`/*new*)

- Suggested replacement for `begindvibox` as a normal named hook. Executed at the very top inside the first shipoutbox.

- Not being implemented as a box register internally also means that it could execute other code than just adding `\specials` etc to the shipout box.

Hook (6): shipout (`\@outputpage`/*new*)

- Executed directly at the top of the shiout box but only for shipouts after the first (which uses `firstshipout`.

- For code that should be executed at all shipouts one need to put it into both `firstshipout` and `shipout`.

The above is probably not general enough in the light of what `atbegshi` implements (fill in correct details ... not done):

Hook (7): shipoutbox (`\@outputpage`/*atbegshi*)

-

Hook (8): forgroundshipoutbox (`\@outputpage`/*atbegshi*)

-

2.2.2 Making columns

Hook (9): pre (`\@makecol`/*new*)

- `manyfoot` and similar packages like to take control at the very beginning of column generation.

Hook (10): post (`\@makecol`/*new*)

- And we may have some post-processing going on after the column is made up.

Config (1): footins (`\@makecol`/*new*)

- To support manipulation of footnote text (like footnotes as paras, in 2 columns etc).

  At this point the assembled footnotes are inside box `\footins` and any manipulation needs to globally write them back in there.

Config (2): blocks (`\@makecol`/*new*)

- A column supports the following block elements:
  - galley text (already inside `\@outputbox`)
  - footnotes (initially inside `\footins`)
  - top and bottom floats (initially inside the LaTeX 2_ε float lists for top and bottom)

  These can appear in different order within the column and in this config point the ordering and any special spacing between them is specified. The final result is stored in `\@outputbox`.

  For specification only the following commands should be used:

  - `\@makecolappendfootnotes`*separation* The separation has to be `\vfill` or `\@makecol@moveskip` (which is not quite the right interface).

- `\@makecolattachfloats`
- `\@makecolappendstuff`*vertical material*

For example to get the footnotes above the bottom floats one would specify:

```
\SetupConfigPoint{@makecol}{blocks}
                 {\@makecol@appendfootnotes {\vfill}%
                  \@makecol@appendfloats}
```

Discussion notes:

- Maybe outputbox should start out empty there should be an explicit `\makecolappendtext`
- Technically attaching top and bottom floats in one go should be enough but perhaps it would be clearer if there are two commands `\@makecolappendtopfloats` and `\@makecolappendbottomfloats` even though their order should always be top above bottom.
- There have to be some clear rules on how `\@makecolappendstuff` can be used.
- Should spaces be automatically suppressed in configuration point setups?

Config (3): splitfootnotemessage (`\@makecol`/*new*)

- Executes if a footnote gets split. By default empty but could be used to set up a warning message or similar in that case.

2.3 Heading commands

Hook (11): before-break (heading/*new*)

- For heading commands it is helpful if one can issue `\mark` commands, e.g. `\PutMark` from `xmarks` to support running header or footer setup. This has to be possible directly after heading title (where it is currently supported through commands like `\sectionmark`) but also directly before the break to determine the placement of the heading (e.g., at the very top of a page/column or elsewhere). The mark needs to come after the heading has done its magic with any preceding space, it can't hide such space from the command. In other words it need to happen basically in the middle of `\addpenalty`.
- For L3 the galley concept might be able to handle this properly, but most likely also need more than one point of code injection.