The `ltmarks.dtx` code*

Frank Mittelbach, LaTeX Project†

August 22, 2024

Abstract

Marks are used to communicate information about the content of a page to the output routine. For example, in order to construct running headers, the output routine needs information about which section names are present on a page, and this information is passed to it through the mark system. However, marks may also be used for other purposes. This module provides a generalized mechanism for marks of independent classes.

Contents

---

*This file has version v1.0g dated 2024/07/08, © LaTeX Project.

†E-mail: latex-team@latex-project.org

1   Introduction

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

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

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

Another problem for LaTeX was that it wanted to use several "independent" marks and in the early implementations of TeX there was only a single `\mark` command available. For that reason LaTeX implemented its own mark mechanism where the marks always contained two parts with their own interfaces: `\markboth` and `\markright` to set marks and `\leftmark` and `\rightmark` to retrieve them.

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

The new mechanism overcomes both issues:

- It provides arbitrarily many, fully independent named marks, that can be allocated and, from that point onwards, used.

- It offers access for each such marks to retrieve its top, first, and bottom values separately.

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

2   Design-level and code-level interfaces

The interfaces are mainly meant for package developers, but they are usable (with appropriate care) also in the document preamble, for example, when setting up special running

headers with `fancyhdr`, etc. They are therefore available both as CamelCase commands as well as commands for use in the L3 programming layer. Both are described together below.

\NewMarkClass	\NewMarkClass {⟨*class*⟩}
\mark_new_class:n	\mark_new_class:n {⟨*class*⟩}

Declares a new ⟨*class*⟩ of marks to be tracked by LaTeX. Each ⟨*class*⟩ must be declared before it is used.

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

\InsertMark	\InsertMark {⟨*class*⟩} {⟨*text*⟩}
\mark_insert:nn	\mark_insert:nn {⟨*class*⟩} {⟨*text*⟩}

Adds a mark to the current galley for the ⟨*class*⟩, containing the ⟨*text*⟩.

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

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

insertmark	\AddToHook {insertmark} {⟨*code*⟩}

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

\TopMark	⋆	\TopMark [⟨*region*⟩] {⟨*class*⟩}
\FirstMark	⋆	\FirstMark [⟨*region*⟩] {⟨*class*⟩}
\LastMark	⋆	\LastMark [⟨*region*⟩] {⟨*class*⟩}
\mark_use_top:nn	⋆	\mark_use_top:nn {⟨*region*⟩} {⟨*class*⟩}
\mark_use_first:nn	⋆	\mark_use_first:nn {⟨*region*⟩} {⟨*class*⟩}
\mark_use_last:nn	⋆	\mark_use_last:nn {⟨*region*⟩} {⟨*class*⟩}

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

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

The "first" and "last" marks are those seen first and last in the current region/page, respectively. The "top" mark is the last mark of the ⟨*class*⟩ seen in an earlier region, i.e., the ⟨*text*⟩ what would be "current" at the very top of the region.

Important! The commands are only meaningful inside the output routine, in other places their result is (while not random) unpredictable due to the way LaTeX cuts text material into pages.

Currently, $\langle region \rangle$ is one of `page`, `previous-page`, `column`, and `previous-column`. If a page has just been finished then the region `page` refers to the current page and `previous-page`, as the name indicates, to the page that has been finished previously. This means you are able to access mark information for the current page as well as for the page before if you are inside the output routine, without the need to explicitly save that information beforehand.

In single column documents the `column` is the same as the `page` region, but in two-column documents, `column` refers to the current column that just got finished and previous-column to the one previously finished. Code for running headers are (in standard LaTeX) only evaluated when both columns are assembled, which is another way of saying that in that case `previous-column` refers to the left column and `column` to the right column. However, to make this a bit nicer to access, there are also alias regions named `first-column` and `last-column`[1] to access these regions.[2]

Note that you can only look backwards at already processed regions, e.g., in a `twoside` document finishing a recto (odd, right-hand) page you can access the data from the facing verso (left-hand) page, but if you are finishing a left-hand page you can't integrate data from the upcoming right-hand page. If such a scenario needs to be realized then it is necessary to save the left-hand page temporarily instead of finalizing it, process material for the right-hand page and once both are ready, attach running headers and footers and shipout out both in one go.[3]

`\IfMarksEqualTF` ⋆	`\IfMarksEqualTF    [`$\langle region \rangle$`] {`$\langle class \rangle$`} {`$\langle pos_1 \rangle$`} {`$\langle pos_2 \rangle$`} {`$\langle true \rangle$`} {`$\langle false \rangle$`}`
`\mark_if_eq:nnnnTF` ⋆	`\mark_if_eq:nnnnTF {`$\langle region \rangle$`} {`$\langle class \rangle$`} {`$\langle pos_1 \rangle$`} {`$\langle pos_2 \rangle$`} {`$\langle true \rangle$`} {`$\langle false \rangle$`}`
`\mark_if_eq:nnnnnnTF` ⋆	`\mark_if_eq:nnnnnnTF {`$\langle region_1 \rangle$`} {`$\langle class_1 \rangle$`} {`$\langle pos_1 \rangle$`}`
	`                    {`$\langle region_2 \rangle$`} {`$\langle class_2 \rangle$`} {`$\langle pos_2 \rangle$`} {`$\langle true \rangle$`} {`$\langle false \rangle$`}`

These conditionals allow you to compare the content of two marks and act based on the result. The commands work in an expansion context, if necessary.

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

The first two conditionals cover only the common scenarios. Both marks are picked up from the same $\langle region \rangle$ (by default `page`) and they have to be of the same $\langle class \rangle$.[4] The $\langle pos_i \rangle$ argument can be either `top`, `first`, or `last`.

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

If you wish to compare marks across different regions or across different classes, you have to do it using the generic test only available in the L3 programming layer or do it manually, i.e., get the marks and then compare the values yourself.[5]

---

[1]This is called "last" not "second" in anticipation of extending the mechanism to multiple columns, where first and last would still make sense.

[2]At the moment there aren't any `previous-...-column` regions to access the columns from the previous page. If necessary, the mechanism could be easily augmented to cover them too, though.

[3]As of now that scenario is not yet officially supported.

[4]If an undeclared mark class is used the tests return *true* (not an error).

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

2.1 Use cases for conditionals

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

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

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

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

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

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

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

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

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

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

2.2 Debugging mark code

`\DebugMarksOn`
`\DebugMarksOff`
`\mark_debug_on:`
`\mark_debug_off:`

`\DebugMarksOn ... \DebugMarksOff`

Commands to turn the debugging of mark code on or off. The debugging output is rather coarse and not really intended for normal use at this point in time.

3 Application examples

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

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

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

to be extended with additional application examples

4 Legacy LaTeX 2ε interface

Here we describe the interfaces that LaTeX 2ε offered since the early nineties and some minor extensions.

4.1 Legacy design-level and document-level interfaces

`\markboth`	`\markboth  {⟨left⟩} {⟨right⟩}`
`\markright`	`\markright {⟨right⟩}`

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

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

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

`\leftmark` ⋆	`\leftmark`
`\rightmark` ⋆	`\rightmark`

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

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

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

4.2 Legacy interface extensions

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

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

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

Thus, if you have a section at the start of a page and you would ask for `\FirstMark{2e-right}` you would get an empty string even if there are subsections on that page. But `2e-right-nonempty` would then give you the first or last subsection on that page. Of course, nothing is simple. If there are no subsections it would tell you the last subsection from an earlier page. We therefore need comparison tools, e.g., if top and first are identical you know that the value is bogus, i.e., a suitable implementation would be

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

5   Notes on the mechanism

In contrast to vanilla TeX, ε-TeX extends the mark system to allow multiple independent marks. However, it does not solve the `\topmark` problem which means that LaTeX still needs to manage marks almost independently of TeX. The reason for this is that the more complex output routine used by LaTeX to handle floats (and related structures) means that `\topmark(s)` remain unreliable. Each time the output routine is fired up, TeX moves `\botmark` to `\topmark`, and while ε-TeX extends this to multiple registers the fundamental concept remains the same. That means that the state of marks needs to be tracked by LaTeX itself. An early implementation of this package used TeX's `\botmark` only to ensure the correct interaction with the output routine (this was before the ε-TeX mechanism was even available). However, other than in a prototype implementation for LaTeX3, this package was never made public.

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

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

- For every type of marks we allocate a mark class so that in the output routine TeX can calculate for each class the current top, first, and bottom mark independently. For this we use `\newmarks`, i.e., one marks register per class.

- As already mentioned firing up an output routine without shipping out a page means that TeX's top marks get wrong so it is impossible to rely on TeX's approach directly. What we do instead is to keep track of the real marks (for the last page or more generally last region) in some global variables.

- These variables are updated in the output routine at defined places, i.e., when we do real output processing but not if we use special output routines to do internal housekeeping.

- The trick we use to get correctly updated variables is the following: the material that contains new marks (for example the page to be shipped out) is stored in a box. We then use TeX primitive box splitting functions by splitting off the largest amount possible (which should be the whole box if nothing goes really wrong). While that seems a rather pointless thing to do, it has one important side effect: TeX sets up first and bottom marks for each mark class from the material it has split off. This way we get the first and last marks (if there have been any) from the material in the box.

- The top marks are simply the last marks from the previous page or region. And if there hasn't been a first or bottom mark in the box then the new top mark also becomes new first and last mark for that class.

- That mark data is then stored in global token lists for use during the output routine and legacy commands such as `\leftmark` or new commands such as `\TopMark` simply access the data stored in these token lists.

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

6   Internal output routine functions

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

```
1 ⟨@@=mark⟩
```

`\__mark_update_singlecol_structures:`   `\__mark_update_singlecol_structures:`

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

`\__mark_update_dblcol_structures:`   `\__mark_update_singlecol_structures:`

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

`\__mark_update_structure_from_material:nn`	`\__mark_update_structure_from_material:nn {⟨region⟩}`
	`{⟨material with marks⟩}`

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

As a consequence, the allowed values for ⟨`region`⟩ are `page` and `column` because only they have `previous-...` counterparts.

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

`\__mark_update_structure_alias:nn`	`\__mark_update_structure_alias:nn {⟨alias⟩} {⟨source⟩}`

Helper function that copies all mark values in the ⟨`source`⟩ region to ⟨`alias`⟩, i.e., make the structures identical. Used to update the `previous-...` structures inside `\__mark_-update_structure_from_material:nn` and `first-column` and `last-column` structures inside `\__mark_update_singlecol_structures:` or `\__mark_update_dblcol_-structures:`.

`\__mark_update_structure_to_err:n`	`\__mark_update_structure_to_err:n {⟨region⟩}`

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

`\__mark_get_marks_for_reinsertion:nNN`	`\__mark_get_marks_for_reinsertion:nNN {⟨source⟩}`
	`⟨token-list-var for collecting first marks⟩`
	`⟨token-list-var for collecting last marks⟩`

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

It collect all the top-level marks from inside the ⟨`source`⟩ and adds suitable `\mark_-insert:nn` in the two token lists. These token lists can then be executed at the right place to reinsert the marks, e.g., directly after the box. This is, for example, used by `multicol` when a short balanced `multicols` is returned to the galley for typesetting.

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

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

7 The Implementation

2 ⟨*2ekernel | latexrelease⟩

9

`\ExplSyntaxOn`

⟨latexrelease⟩`\NewModuleRelease{2022/06/01}{ltmarks}`

⟨latexrelease⟩   `{Marks~handling}`

7.1 Allocating new mark classes

`\g__mark_classes_seq` A list holding all the mark classes that have been declared.

`\seq_new:N \g__mark_classes_seq`

`\mark_new_class:n` A mark class is created by initializing a number of data structures. First, we get a register
`\__mark_new_class:nn` number to refer to the mark class. The new mark class is then added to the `\g__mark_-`
`classes_seq` sequence to be able to easily loop over all classes. Finally a number of
top-level global token lists are declared that hold various versions of the mark for access.

`\cs_new_protected:Npn \mark_new_class:n #1`
`{`
`  \seq_if_in:NnTF \g__mark_classes_seq {#1}`
`    {`
`      \msg_error:nnn { mark } { class-already-defined }`
`        {#1}`
`    }`
`    { \__mark_new_class:nn {#1} }`
`}`

This is only available in the preamble.

`\@onlypreamble \mark_new_class:n`

The internal command carries out the necessary allocations.

`\cs_new_protected:Npn \__mark_new_class:nn #1`
`{`
⟨∗trace⟩
`  \__mark_debug:n { \iow_term:x { Marks:~new~mark:~#1~\msg_line_context: } }`
⟨/trace⟩

Use the LaTeX 2_ε interface for now as the L3 programming layer doesn't have one for
marks yet.

`  \exp_args:Nc \newmarks {c__mark_class_ #1 _mark}`

Remember the new class in the sequence.

`  \seq_gput_right:Nn \g__mark_classes_seq {#1}`

We need three token lists for each region, one for top, first, and last.

`  \tl_new:c   { g__mark_page_top_    #1 _tl }`
`  \tl_new:c   { g__mark_page_first_  #1 _tl }`
`  \tl_new:c   { g__mark_page_last_   #1 _tl }`

For the `page` region we also keep track of the `previous-page`.

`  \tl_new:c   { g__mark_previous-page_top_    #1 _tl }`
`  \tl_new:c   { g__mark_previous-page_first_  #1 _tl }`
`  \tl_new:c   { g__mark_previous-page_last_   #1 _tl }`

Same game for `column` and `previous-column`

```
30    \tl_new:c    { g__mark_column_top_    #1 _tl }
31    \tl_new:c    { g__mark_column_first_ #1 _tl }
32    \tl_new:c    { g__mark_column_last_  #1 _tl }
33    \tl_new:c    { g__mark_previous-column_top_    #1 _tl }
34    \tl_new:c    { g__mark_previous-column_first_ #1 _tl }
35    \tl_new:c    { g__mark_previous-column_last_  #1 _tl }
```

But for columns we also allocate token lists for the alias regions `first-column` and `last-column`.

```
36    \tl_new:c    { g__mark_first-column_top_    #1 _tl }
37    \tl_new:c    { g__mark_first-column_first_ #1 _tl }
38    \tl_new:c    { g__mark_first-column_last_  #1 _tl }
39    \tl_new:c    { g__mark_last-column_top_     #1 _tl }
40    \tl_new:c    { g__mark_last-column_first_  #1 _tl }
41    \tl_new:c    { g__mark_last-column_last_   #1 _tl }
```

All marks will have an identification at the beginning of the form `\__mark_id:n{`⟨*number*⟩`}` and therefore the initial empty values should have that too, so that data extraction is going to be uniform.

```
42    \tl_set:cn    { g__mark_page_top_    #1 _tl }{ \__mark_id:n{0} }
43    \tl_set:cn    { g__mark_page_first_ #1 _tl }{ \__mark_id:n{0} }
44    \tl_set:cn    { g__mark_page_last_  #1 _tl }{ \__mark_id:n{0} }
45    \tl_set:cn    { g__mark_previous-page_top_    #1 _tl }{ \__mark_id:n{0} }
46    \tl_set:cn    { g__mark_previous-page_first_ #1 _tl }{ \__mark_id:n{0} }
47    \tl_set:cn    { g__mark_previous-page_last_  #1 _tl }{ \__mark_id:n{0} }
48    \tl_set:cn    { g__mark_column_top_    #1 _tl }{ \__mark_id:n{0} }
49    \tl_set:cn    { g__mark_column_first_ #1 _tl }{ \__mark_id:n{0} }
50    \tl_set:cn    { g__mark_column_last_  #1 _tl }{ \__mark_id:n{0} }
51    \tl_set:cn    { g__mark_previous-column_top_    #1 _tl }{ \__mark_id:n{0} }
52    \tl_set:cn    { g__mark_previous-column_first_ #1 _tl }{ \__mark_id:n{0} }
53    \tl_set:cn    { g__mark_previous-column_last_  #1 _tl }{ \__mark_id:n{0} }
54    \tl_set:cn    { g__mark_first-column_top_    #1 _tl }{ \__mark_id:n{0} }
55    \tl_set:cn    { g__mark_first-column_first_ #1 _tl }{ \__mark_id:n{0} }
56    \tl_set:cn    { g__mark_first-column_last_  #1 _tl }{ \__mark_id:n{0} }
57    \tl_set:cn    { g__mark_last-column_top_     #1 _tl }{ \__mark_id:n{0} }
58    \tl_set:cn    { g__mark_last-column_first_  #1 _tl }{ \__mark_id:n{0} }
59    \tl_set:cn    { g__mark_last-column_last_   #1 _tl }{ \__mark_id:n{0} }
60    }
```

(*End of definition for* `\mark_new_class:n` *and* `\__mark_new_class:nn`*. This function is documented on page* *.*)

7.2 Updating mark structures

`\l__mark_box`
`\l__mark_ii_box`
`\g__mark_tmp_tl`
`\g__mark_new_top_tl`

For some operations we need two temporary private boxes and two private global token lists.

```
61    \box_new:N \l__mark_box
62    \box_new:N \l__mark_ii_box
63    \tl_new:N  \g__mark_tmp_tl
64    \tl_new:N  \g__mark_new_top_tl
```

(*End of definition for* `\l__mark_box` *and others.*)

__mark_extract_and_handle_marks:nn This is the main macro to extract and handle marks inside some vertical material. It is used by `\__mark_update_structure_from_material:nn` (for updating the mark structure for a region based on the marks found) and by `\__mark_get_marks_for_-reinsertion:nNN` (for extracting marks from some material and prepare for reinserting them later (e.g., out of a box that is placed as a box into the main galley).

```
65   \cs_new_protected:Npn \__mark_extract_and_handle_marks:nn #1#2 {
```

This macro expects code to handle extracted marks in its first argument and vertical material (not boxed or just consisting of a single vertical box) as its second. It extracts top-level mark information from #2, stores them as split marks and then calls #1 to make use of this information.

If it finds a forced break in the material it removes it and then restarts the attempt without it.

We start with a group to keep most changes local.

```
66        \group_begin:
```

Getting the first and last marks out of the material in #2 is done by putting the material in a box and then doing a split operation to the maximum size possible (which hopefully gets us all of the content).[6] Because this action is used only to get the mark values, we don't want any underfull box warnings so we (locally) turn those off.

```
67        \dim_set_eq:NN \tex_splitmaxdepth:D \c_max_dim
68        \int_set_eq:NN \tex_vbadness:D       \c_max_int
69        \dim_set_eq:NN \tex_vfuzz:D          \c_max_dim
```

There is a further complication: if the material contains infinite shrinking glue then a `\vsplit` operation will balk with a low-level error. Now pages or columns, which are our main concern here, can't have such infinite shrinkage if they are cut straight from the galley, however the use of `\enlargethispage` actually does add some at the very bottom (and also wraps the whole page into a box by itself, so if we leave it this way then a) we get this error and b) we don't see any marks because they are hidden one level down).

Another possible issue are packages or user code that place stray `\vbox`es directly into the main galley (an example is marginnote that attaches its marginals in this way). If such boxes end up as the last item on the page we should not unpack them.

All these issues need to be handled, which is done in `\__mark_prepare_and_-extract:nn`.

```
70        \__mark_prepare_and_extract:nn {#1} {#2}
```

Once all mark classes have been processed, the data structures are updated and we can close the group, which undoes our local changes and retains only the global ones.

```
71        \group_end:
72   }
```

(*End of definition for* `\__mark_extract_and_handle_marks:nn`.)

__mark_prepare_and_extract:nn This macro does the dirty work. It is not directly integrated in `\__mark_extract_and_-handle_marks:nn` because we may have to call it recursively if we find forced breaks.

```
73   \cs_new_protected:Npn \__mark_prepare_and_extract:nn #1#2 {
```

---

[6]With normal column material cut from the main galley we should always get all material in one go, but in certain situations, for example, in a multicols environment that contains some `\columnbreak`s a single split operation will not be enough. Thus, this is something we need to handle.

To handle the \enlargethispage case we do an \unskip to get rid of any glue that is present at the very end of the material and also check if we have then a \vbox as the last item and if so unpack that too, but only under certain conditions, see below. All this is temporary done in a group, just for getting the marks out, so it doesn't affect the final page production.

```
74    \vbox_set:Nn \l__mark_box
75      {
76        #2
77        \tex_unskip:D
78        \box_set_to_last:N \l__mark_box
```

After having removed the last box from the current list (if there was one) we check whether the vertical list is now empty. If not, then the last box is definitely not the one from \enlargethispage and so we can, and should, leave it alone. Otherwise we check if this last box is a \vbox.

```
79        \int_compare:nNnT \tex_lastnodetype:D < 0
80          {
81            \box_if_vertical:NT \l__mark_box
```

If it is, we unpack the box.

```
82              { \vbox_unpack:N \l__mark_box }
83          }
```

If it wasn't a vbox, it was either an hbox or there was no box. Given that we are only interested in the marks we don't need put it back in that case.

```
84      }
```

We are now ready to \vsplit the box to get at the marks. If the box contains some infinite negative glue the TEX will produce an error complaining about it but it will correctly find the the split marks. Given that we can't prevent that error, we hide it from the user and ensure that TEX doesn't stop. The error message still shows in the log, but even that is mitigated as best as possible—see the definition of __mark_vbox_-set_split_to_maxdimen:NN for the tricks employed.

```
85      \__mark_vbox_set_split_to_maxdimen:NN \l__mark_ii_box \l__mark_box
```

After splitting we check if there is anything left in \l__mark_box. If not then the above split has set some split marks that we can then use to finish the extraction:

```
86      \box_if_empty:NTF \l__mark_box
87        { #1 }
```

If we have a remainder after the split then this means that there was some forced break in the material. We get rid of that by combining the content of the two boxes and restart.

```
88        {
89 ⟨∗trace⟩
90          \__mark_debug:n { \iow_term:x
91            { Marks:~ mark~ extraction~needs~ recursion~
92              \msg_line_context: } }
93 ⟨/trace⟩
94          \__mark_prepare_and_extract:nn {#1}
95            { \vbox_unpack:N \l__mark_ii_box
96              \vbox_unpack:N \l__mark_box    }
97        }
98 }
```

(*End of definition for* __mark_prepare_and_extract:nn.)

13

`\__mark_vbox_set_split_to_maxdimen:NN` Split a box to get at its marks without pausing even if TeX is producing an error message because of infinite negative glue in the box. If there is such an error we ensure that it only shows up in the log but not on the terminal.

The nice low-level hack by DPC records in the `.log` that a glue shrinkage error is harmless.

We disguise `\c_max_dim` in an odd looking csname, which then shows up as part of the display of an error message if that error happens. This csname forms part of the error display so what you get is something like

```
! Infinite glue shrinkage found in box being split.
<argument> Infinite shrink error above ignored !
l. ...  }
```

which hopefully makes it clear that the error is harmless and and should be ignored by the reader of the `.log`.

```
99 \cs_set_eq:cN {Infinite~shrink~error~above~ignored~!}\c_max_dim
```

The whole definition of `\__mark_vbox_set_split_to_maxdimen:NN` below is fully expanded, so we have to use a lot of `\exp_not:N` commands to prevent expansion where necessary.

```
100 \cs_new_protected:Npx \__mark_vbox_set_split_to_maxdimen:NN #1#2 {
```

We start by saving the current interaction and escape char settings.

```
101   \tl_set:Ne \exp_not:N \l__mark_saved_parameters_tl
102     {
103       \tex_interactionmode:D
104         \exp_not:N \int_use:N \tex_interactionmode:D \scan_stop:
105       \tex_escapechar:D
106         \exp_not:N \int_use:N \tex_escapechar:D \scan_stop:
107     }
```

Then we change them so that no escape char is printed in the error message (accounts for the missing backslash in front of `Infinite shrink ...`) and we set the interaction to `\nonstopmode` so that the the error (if any) just goes into the `.log` file and TeX doesn't stop at that point.

```
108   \tex_escapechar:D    -1 \scan_stop:
109   \tex_interactionmode:D 0 \scan_stop:
```

Then we do the splitting of the box to `\c_max_dim` to get at the marks. This may generate the error we are worried about, i.e., if the box contains infinite negative glue. However, TeX makes this glue finite and continues, which means we get our split marks which is really all we care about.

```
110   \tex_setbox:D #1 \tex_vsplit:D #2 to
```

The `\use:n` may seem pointless, and it is to some extent, but we need it to get our disguised `\c_max_dim` displayed properly as part of the error message if there is one. Without it, the display would show only part of what we want it to show (try it).

```
111     \exp_not:N \use:n {
112       \use:c{Infinite~shrink~error~above~ignored~!}
113     }
```

Finally, we change the escape char and the interaction mode back to what it was before:

```
114   \exp_not:N \l__mark_saved_parameters_tl
115 }
```

(End of definition for `\__mark_vbox_set_split_to_maxdimen:NN`.*)*

`\l__mark_saved_parameters_tl` The temporary variable used for resetting escape char and interaction mode.

```
116 \tl_new:N \l__mark_saved_parameters_tl
```

(End of definition for `\l__mark_saved_parameters_tl`.*)*

`\__mark_update_structure_from_material:nn` This function updates the mark structures of a region. The first argument is the region to update and second argument receives the material that holds the marks. Out of this material we extract the first and last marks for all classes (if there are any) to do the assignments.

```
117 \cs_new_protected:Npn \__mark_update_structure_from_material:nn #1#2 {
118     \__mark_extract_and_handle_marks:nn
```

Once the marks can be extracted we update the structure from the split marks (code in `\__mark_update_structure_from_splitmarks:n`).

```
119         { \__mark_update_structure_from_splitmarks:n {#1} }
120         { #2 }
121 }
```

(End of definition for `\__mark_update_structure_from_material:nn`.*)*

`\__mark_update_structure_from_splitmarks:n` This macro is called after we have done a `\tex_vsplit:D` operation and the mark data is in the split marks.

```
122 \cs_new_protected:Npn \__mark_update_structure_from_splitmarks:n #1 {
```

The first thing we do is to copy the current region structure to `previous-...`; this leaves the current structure untouched so we can update it class by class (which is necessary).

```
123     \__mark_update_structure_alias:nn { previous-#1 } {#1}
```

After this action we can get first and last marks of the various classes through `\tex_-splitfirstmarks:D` and `\tex_splitbotmarks:D`. So now we loop over all classes stored in `\g__mark_classes_seq`.

```
124     \seq_map_inline:Nn \g__mark_classes_seq
125             {
```

First action: get the last mark from the previous region, i.e., `previous-#1`. But because it is also still inside `#1`, at the moment we use that to construct the name because this is a tiny bit faster. Given that we need this value in various assignments we store it away which avoids unnecessary further csname generations.

```
126                 \tl_gset_eq:Nc \g__mark_new_top_tl { g__mark_#1_last_##1_tl }
```

This will first of all become the new top mark for the current class.

```
127                 \tl_gset_eq:cN { g__mark_#1_top_##1_tl } \g__mark_new_top_tl
```

Next action is to get ourselves the new last mark from the material supplied.

```
128                 \tl_gset:No \g__mark_tmp_tl
129                   { \tex_splitbotmarks:D \use:c { c__mark_class_##1_mark } }
```

15

If this mark doesn't exist then obviously neither does the first mark, so both become the last mark from the previous region. We have to be a little careful here: something like `\mark_insert:nn{foo}{}` adds an "empty" mark that should not be confused with no mark at all. But no mark in our material will result in `\g__mark_tmp_tl` being fully empty. This is why we have to make sure that "empty" from `\mark_insert:nn` only appears to be empty when typeset but fails the next test (see below how this is done).

```
130                    \tl_if_empty:NTF \g__mark_tmp_tl
131                      {
132                        \tl_gset_eq:cN { g__mark_#1_last_ ##1_tl }
133                          \g__mark_new_top_tl
134                        \tl_gset_eq:cN { g__mark_#1_first_##1_tl }
135                          \g__mark_new_top_tl
136                      }
```

If it wasn't empty, i.e., if it had a real value then we use this value for our new last mark instead.

```
137                      {
138                        \tl_gset_eq:cN { g__mark_#1_last_##1_tl } \g__mark_tmp_tl
```

Because we had a last mark we also have a first mark (which might be the same, but might be not), so we pick that up and assign it to the appropriate token list. This explains why we first checked for the last mark because that makes the processing faster in case there is none.

```
139                        \tl_gset:co { g__mark_#1_first_##1_tl }
140                          {
141                            \tex_splitfirstmarks:D
142                              \use:c { c__mark_class_##1_mark }
143                          }
144                      }
145                  }
146 }
```

(*End of definition for* `\__mark_update_structure_from_splitmarks:n`.)

`\__mark_get_marks_for_reinsertion:nNN` This function extracts the marks from the material in the first argument but it does not update any the mark structures. Instead, it collects the marks in the token lists given as the second and third argument, in such a way that they can be reinserted by just executing the token lists.[7]

```
147 \cs_new_protected:Npn \__mark_get_marks_for_reinsertion:nNN #1#2#3 {
```

First we clear the temporary token lists as we haven't seen any marks yet.

```
148    \tl_clear:N \g__mark_first_marks_tl
149    \tl_clear:N \g__mark_last_marks_tl
```

Then we extract all top-level marks, thereby filling the token lists with suitable `\mark_insert:nn` calls.

```
150    \__mark_extract_and_handle_marks:nn
```

The first argument holds the code for fill the token lists and the second is the material we extract from.

```
151        \__mark_get_from_splitmarks:
152        { #1 }
```

---

[7]It is probably enough to collect everything in a single token list as long as we put the first marks first and the last marks last). But for extra flexibility, I currently use 2 token lists. This might change when it is really clear that this is never needed.

Finally, we copy the updated (or not updated) temporary token lists to the two that have been supplied when the function was called. By convention "get" operations return their values in local variables and `\__mark_extract_and_handle_marks:nn` runs in a group, which is why we have to use global temporary variables for collecting.

```
153    \tl_set_eq:NN #2 \g__mark_first_marks_tl
154    \tl_set_eq:NN #3 \g__mark_last_marks_tl
155 }
```

(*End of definition for* `\__mark_get_marks_for_reinsertion:nNN`.)

`\__mark_get_from_splitmarks:`   This function is called after we have done a `\vsplit` to update the split marks. It loops through all mark classes to find out if there are marks for this class and if so updates the global tls used for collecting.

```
156 \cs_new_protected:Npn  \__mark_get_from_splitmarks:  {
157    \seq_map_inline:Nn \g__mark_classes_seq
158            {
```

First we to get the last mark for the current class from the material supplied.

```
159                \tl_gset:No \g__mark_tmp_tl
160                { \tex_splitbotmarks:D \use:c { c__mark_class_##1_mark } }
```

If this mark doesn't exist then obviously first mark doesn't either, so we do nothing (other than issuing some debugging info).

We have to be a little careful here: something like `\mark_insert:nn{foo}{}` adds an "empty" mark that we should not confuse with the case where there is no mark at all.

When there is no mark at all we get a truly empty `\g__mark_tmp_tl` as a result. This is why we have to make sure that an "empty" mark generated with `\mark_insert:nn` only appears to be empty when it is typeset, but fails the next test (see below how this is done).

```
161                \tl_if_empty:NTF \g__mark_tmp_tl
162                  {
163 ⟨*trace⟩
164                      \__mark_debug:n { \iow_term:x { Marks:~no~ marks~
165                          for~ class~ '##1'~\msg_line_context: } }
166 ⟨/trace⟩
167                  }
```

If it wasn't empty, i.e., if it had a real value then we use this value for our new last mark instead. This means we put an appropriate `\mark_insert:nn` statement into `\g__mark_last_marks_tl`.

```
168                  {
169 ⟨*trace⟩
170                      \__mark_debug:n { \iow_term:x { Marks:~ extract~ last~
171                          mark~ for~ class~ '##1'~ =~  \g__mark_tmp_tl } }
172 ⟨/trace⟩
173                      \tl_gput_right:Ne \g__mark_last_marks_tl
174                        { \mark_insert:nn {##1} { \g__mark_tmp_tl } } }
```

Because we had a last mark we also have a first mark (which might be the same, but might not be), so we pick that up and add it to the `\g__mark_first_marks_tl` token list. This explains why we first checked for the last mark because that makes the processing faster in case there is none.

```
175 ⟨*trace⟩
176                      \__mark_debug:n { \iow_term:x { Marks:~ extract~ first~
```

```
177                              mark~ for~ class~ '##1'~ =~
178                                \tex_splitfirstmarks:D
179                                  \use:c { c__mark_class_##1_mark }
180                          } }
181 ⟨/trace⟩
182                        \tl_gput_right:Ne \g__mark_first_marks_tl
183                          { \mark_insert:nn {##1}
184                            {
185                              \tex_splitfirstmarks:D
186                              \use:c { c__mark_class_##1_mark }
187                            }
188                          }
189                      }
190                  }
191 }
```

(End of definition for `\__mark_get_from_splitmarks:`*.)*

<div>

`\g__mark_first_marks_tl`
`\g__mark_last_marks_tl`

These are two global temporary variables used in the code above.

```
192 \tl_new:N  \g__mark_first_marks_tl
193 \tl_new:N  \g__mark_last_marks_tl
```

(End of definition for `\g__mark_first_marks_tl` *and* `\g__mark_last_marks_tl`*.)*

</div>

<div>

`\__mark_update_structure_alias:nn`

This function copies the structure for one region to another (name), e.g., from `page` to `previous-page` above, or later from `column` to `first-column`, etc.

```
194 \cs_new_protected:Npn \__mark_update_structure_alias:nn #1#2 {
```

This requires a simple loop through all mark classes copying the token list from one name to the next.

```
195   \seq_map_inline:Nn \g__mark_classes_seq
196     {
197       \tl_gset_eq:cc { g__mark_ #1 _top_   ##1 _tl }
198                      { g__mark_ #2 _top_   ##1 _tl }
199       \tl_gset_eq:cc { g__mark_ #1 _first_ ##1 _tl }
200                      { g__mark_ #2 _first_ ##1 _tl }
201       \tl_gset_eq:cc { g__mark_ #1 _last_  ##1 _tl }
202                      { g__mark_ #2 _last_  ##1 _tl }
203     }
204 }
```

(End of definition for `\__mark_update_structure_alias:nn`*.)*

</div>

<div>

`\__mark_update_structure_to_err:n`
`\__mark_error:n`

A slight variation is to install a fixed error message as the value.

```
205 \cs_new_protected:Npn \__mark_update_structure_to_err:n #1 {
206   \seq_map_inline:Nn \g__mark_classes_seq
207     {
208       \tl_gset:cn { g__mark_ #1 _top_   ##1 _tl } { \__mark_error:n {#1} }
209       \tl_gset:cn { g__mark_ #1 _first_ ##1 _tl } { \__mark_error:n {#1} }
210       \tl_gset:cn { g__mark_ #1 _last_  ##1 _tl } { \__mark_error:n {#1} }
211     }
212 }
```

</div>

Given that this is used in only one place, we could hardwire the argument which would be a bit more compact, but who knows, perhaps we end up with another reason to use this error command elsewhere, so for now we keep the argument.

```
213 \cs_new_protected:Npn \__mark_error:n #1 {
214     \msg_error:nnn { mark } { invalid-use } {#1}
215 }
```

(*End of definition for* __mark_update_structure_to_err:n *and* __mark_error:n.)

7.3   Placing and retrieving marks

\mark_insert:nn   This function puts a mark for some ⟨*class*⟩ at the current point.

```
216 \cs_new_protected:Npn \mark_insert:nn #1#2
217 {
218     \seq_if_in:NnTF \g__mark_classes_seq {#1}
219         {
```

We need to pass the evaluated argument into the mark but protected commands should not expand including those protected using the \protect approach of LATEX 2ε. We also disable \label and the like.[8]

At this point the code eventually should get a public (and a kernel) hook instead of a set of hardwired settings.

```
220             \group_begin:
```

Within the group we alter some comments, e.g, \label or \index, to do the right at this point. This is done in the kernel hook \@kernel@before@insertmark which is followed by the public hook insertmark that can be used by packages to augment or alter that setup as necessary.

```
221             \@kernel@before@insertmark
222             \hook_use:n { insertmark }
223             \unrestored@protected@xdef \g__mark_tmp_tl
224                 {
```

To ensure that marks are unique we insert a hidden sequence marker at the beginning of the content of the mark containing the sequence number of the mark.

```
225                     \__mark_id:n{ \int_use:N\g__mark_int }
226                     #2
227                 }
228 ⟨*trace⟩
229             \__mark_debug:n{ \iow_term:x { Marks:~ set~#1~<-~
230                 '\tl_to_str:V \g__mark_tmp_tl' ~ \msg_line_context: } }
231 ⟨/trace⟩
232             \tex_marks:D \use:c { c__mark_class_ #1 _mark }
233                 {
```

Here is the trick to avoid truly empty marks: if the result from the above processing is empty we add something which eventually becomes empty, but not immediately; otherwise we just put \g__mark_tmp_tl in.

```
234 % this is no longer needed with 1.0f
235 %               \tl_if_empty:NTF \g__mark_tmp_tl
236 %                   { \exp_not:n { \prg_do_nothing: } }
237 %                   { \exp_not:o { \g__mark_tmp_tl } }
```

---

[8]Straight copy from `latex.ltx` but is this even correct? At least a label in a running header makes little sense if it get set several times! Maybe that needs looking at in the 2e kernel.

```
238            \exp_not:o { \g__mark_tmp_tl }
239          }
240        \group_end:
```

A mark introduces a possible break point and in certain situations that should not happen in vertical mode in LaTeX. This needs some cleanup . . . .

```
241          \if@nobreak\ifvmode\nobreak\fi\fi
242        }
```

If the mark class was not known, raise an error.

```
243      {
244        \msg_error:nnx { mark } { unknown-class }
245          { \tl_to_str:n {#1} }
246      }
247 }
```

(*End of definition for* \mark_insert:nn*. This function is documented on page 3.*)

__mark_id:n A hidden marker is placed into every mark added by \mark_insert:nn. It will will not show up in the output but its argument (a counter value that is incremented) makes all marks unique so the test for "equal" is not fooled by two different marks having the same mark text.

```
248 \cs_new_protected:Npn \__mark_id:n #1 { }
```

(*End of definition for* __mark_id:n*.*)

\@kernel@before@insertmark By default \label, \index, and \glossary do nothing when the mark is inserted.
insertmark
```
249 \int_new:N \g__mark_int
250 \cs_new:Npn \@kernel@before@insertmark {
251        \cs_set_eq:NN \label    \scan_stop:
252        \cs_set_eq:NN \index    \scan_stop:
253        \cs_set_eq:NN \glossary \scan_stop:
```

We count each mark and use that to place a hidden marker in front of the mark text. To ensure that there is no overflow (very unlikely but you never know) we restart every 100000 marks. Thus, if somebody puts more than that number of marks on a single page you could construct a scenario in which that approach fails.

```
254        \int_compare:nNnTF \g__mark_int < {99999}
255            { \int_gincr:N \g__mark_int }
256            { \int_gzero:N \g__mark_int }
257
258 }
```

The public hook to augment the setup.

```
259 \hook_new:n {insertmark}
```

(*End of definition for* \@kernel@before@insertmark *and* insertmark*.*)

\mark_use_top:nn   To retrieve the first, last or top region mark, we grab the appropriate value stored
\mark_use_first:nn in the corresponding token list variable and pass its contents back. These functions
\mark_use_last:nn should be used only in output routines and only after __mark_update_structure_-
from_material:nn has acted, otherwise their value will be wrong.

  If used with an unknown class or region they generate an error (fairly low-level because we are in an expandable context).

Each mark starts with an id and while the id does not print it is nevertheless better to remove it when returning the mark, so that downstream manipulation of the data doesn't have to deal with it.

```
260 \cs_new:Npn \mark_use_first:nn #1#2 { \__mark_use:v { g__mark_#1_first_#2_tl } }
261 \cs_new:Npn \mark_use_last:nn #1#2  { \__mark_use:v { g__mark_#1_last_#2_tl }  }
262 \cs_new:Npn \mark_use_top:nn #1#2   { \__mark_use:v { g__mark_#1_top_#2_tl }   }
```

This is what the `\use_none:nn` accomplishes.

```
263 \cs_new:Npn \__mark_use:n #1 { \exp_not:o { \use_none:nn #1 } }
264 \cs_generate_variant:Nn \__mark_use:n { v }
```

(*End of definition for* `\mark_use_top:nn`*,* `\mark_use_first:nn`*, and* `\mark_use_last:nn`*. These functions are documented on page 3.*)

7.4   Comparing mark values

\mark_if_eq:nnnn*TF*
\mark_if_eq:nnnnnn*TF*

Test if in a given region (`#1`) for a given class (`#2`) the marks in position `#3` and `#4` (top, first, or last) are identical

```
265 \prg_new_conditional:Npnn \mark_if_eq:nnnn #1#2#3#4 { T , F , TF }
266 {
267   \tl_if_eq:ccTF { g__mark_ #1 _#3_ #2 _tl }
268                  { g__mark_ #1 _#4_ #2 _tl }
269                  \prg_return_true:
270                  \prg_return_false:
271 }
```

The fully general test (with two triplets of the form ⟨*region*⟩, ⟨*class*⟩, and ⟨*position*⟩) is this:

```
272 \prg_new_conditional:Npnn \mark_if_eq:nnnnnn #1#2#3#4#5#6 { T , F , TF }
273 {
274   \tl_if_eq:ccTF { g__mark_ #1 _#3_ #2 _tl }
275                  { g__mark_ #4 _#6_ #5 _tl }
276                  \prg_return_true:
277                  \prg_return_false:
278 }
```

(*End of definition for* `\mark_if_eq:nnnnTF` *and* `\mark_if_eq:nnnnnnTF`*. These functions are documented on page 4.*)

7.5   Messages

Mark errors are LaTeX kernel errors:

```
279 \prop_gput:Nnn \g_msg_module_type_prop { mark } { LaTeX }

280 \msg_new:nnnn { mark } { class-already-defined }
281   { Mark~class~'#1'~already~defined }
282   {
283     \c__msg_coding_error_text_tl
284     LaTeX~was~asked~to~define~a~new~mark~class~called~'#1':~
285     this~mark~class~already~exists.
286     \c__msg_return_text_tl
287   }
```

```
288 \msg_new:nnnn { mark } { unknown-class }
289   { Unknown~mark~class~'#1'. }
290   {
291     \c__msg_coding_error_text_tl
292     LaTeX~was~asked~to~manipulate~a~mark~of~class~'#1',~
293     but~this~class~of~marks~does~not~exist.
294   }
295
296 \msg_new:nnnn { mark } { invalid-use }
297   { Mark~region~'#1'~not ~usable }
298   {
299     \c__msg_coding_error_text_tl
300     The~region~'#1'~can~only~be~used~after~
301     all~columns~have~been~assembled.
302     \c__msg_return_text_tl
303   }
```

7.6 Debugging the mark structures

Code and commands in this section are not final, it needs more experimentation to see what kind of tracing information is going to be useful in practice. For now the tracing is mainly meant to be used for code testing and not so much for application testing.

It is quite likely that the commands and the behavior of the tracing might change in the future once we gained some experience with it.

`\g__mark_debug_bool`   Holds the current debugging state.

```
304 \bool_new:N \g__mark_debug_bool
```

(*End of definition for* `\g__mark_debug_bool`.)

`\mark_debug_on:`   Turns debugging on and off by redefining `\__mark_debug:n`.
`\mark_debug_off:`
`\__mark_debug:n`
`\__mark_debug_gset:`

```
305 \cs_new_eq:NN \__mark_debug:n \use_none:n
306 \cs_new_protected:Npn \mark_debug_on:
307   {
308     \bool_gset_true:N \g__mark_debug_bool
309     \__mark_debug_gset:
310   }
311 \cs_new_protected:Npn \mark_debug_off:
312   {
313     \bool_gset_false:N \g__mark_debug_bool
314     \__mark_debug_gset:
315   }
316 \cs_new_protected:Npn \__mark_debug_gset:
317   {
318     \cs_gset_protected:Npx \__mark_debug:n ##1
319       { \bool_if:NT \g__mark_debug_bool {##1} }
320   }
```

(*End of definition for* `\mark_debug_on:` *and others. These functions are documented on page* *5.*)

`\DebugMarksOn`   CamelCase commands for debugging.
`\DebugMarksOff`

```
321 \cs_new_eq:NN \DebugMarksOn    \mark_debug_on:
322 \cs_new_eq:NN \DebugMarksOff   \mark_debug_off:
```

(*End of definition for* \DebugMarksOn *and* \DebugMarksOff. *These functions are documented on page* 5.)

__mark_class_status:nn   Shows the mark values across all regions for one mark class (#2). The first argument gives some ⟨info⟩ to help identifying where the command was called.

```
323 ⟨*trace⟩
324 \cs_new_protected:Npn \__mark_class_status:nn #1#2
325   {
326         \typeout{ Marks:~#2~ #1:}
327         \typeout{\@spaces page~ (current):
328           | \exp_not:v { g__mark_page_top_ #2 _tl }
329           | \exp_not:v { g__mark_page_first_ #2 _tl }
330           | \exp_not:v { g__mark_page_last_ #2 _tl }      |}
331         \typeout{\@spaces page~ (previous):
332           | \exp_not:v { g__mark_previous-page_top_ #2 _tl }
333           | \exp_not:v { g__mark_previous-page_first_ #2 _tl }
334           | \exp_not:v { g__mark_previous-page_last_ #2 _tl }      |}
335         \typeout{\@spaces column~ (previous):
336           | \exp_not:v { g__mark_previous-column_top_ #2 _tl }
337           | \exp_not:v { g__mark_previous-column_first_ #2 _tl }
338           | \exp_not:v { g__mark_previous-column_last_ #2 _tl }      |}
339         \typeout{\@spaces column~ (current):
340           | \exp_not:v { g__mark_column_top_ #2 _tl }
341           | \exp_not:v { g__mark_column_first_ #2 _tl }
342           | \exp_not:v { g__mark_column_last_ #2 _tl }      |}
343         \typeout{\@spaces column~ (first):
344           | \exp_not:v { g__mark_first-column_top_ #2 _tl }
345           | \exp_not:v { g__mark_first-column_first_ #2 _tl }
346           | \exp_not:v { g__mark_first-column_last_ #2 _tl }      |}
347         \typeout{\@spaces column~ (second):
348           | \exp_not:v { g__mark_last-column_top_ #2 _tl }
349           | \exp_not:v { g__mark_last-column_first_ #2 _tl }
350           | \exp_not:v { g__mark_last-column_last_ #2 _tl }      |}
351   }
```

(*End of definition for* __mark_class_status:nn.)

__mark_status:n   Show a snapshot of all mark class values across all regions.

```
352 \cs_new_protected:Npn \__mark_status:n #1
353   {
354     \seq_map_inline:Nn \g__mark_classes_seq
355       { \__mark_class_status:nn {#1} {##1} }
356   }
357 ⟨/trace⟩
```

(*End of definition for* __mark_status:n.)

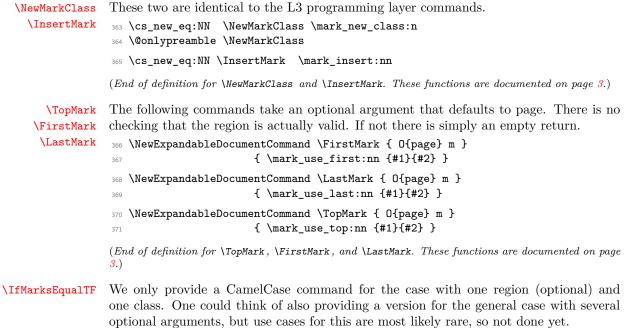\ShowMarksAt   Debugging helper that displays a snapshot of all known mark structures. The argument is a text string that is displayed to help identifying when the snapshot was made.

This may not stay like this (or at all), which is why it isn't yet documented as an official command.

```
358 \cs_new_protected:Npn \ShowMarksAt #1 {
359 ⟨*trace⟩
360     \__mark_debug:n { \__mark_status:n {#1} }
361 ⟨/trace⟩
362 }
```

23

(End of definition for `\ShowMarksAt`*. This function is documented on page* **??***.)*

7.7 Designer-level interfaces

`\NewMarkClass`
`\InsertMark`
These two are identical to the L3 programming layer commands.

```
363 \cs_new_eq:NN  \NewMarkClass \mark_new_class:n
364 \@onlypreamble \NewMarkClass
```

```
365 \cs_new_eq:NN \InsertMark  \mark_insert:nn
```

(End of definition for `\NewMarkClass` *and* `\InsertMark`*. These functions are documented on page* *3.)*

`\TopMark`
`\FirstMark`
`\LastMark`
The following commands take an optional argument that defaults to page. There is no checking that the region is actually valid. If not there is simply an empty return.

```
366 \NewExpandableDocumentCommand \FirstMark { O{page} m }
367                     { \mark_use_first:nn {#1}{#2} }
368 \NewExpandableDocumentCommand \LastMark { O{page} m }
369                     { \mark_use_last:nn {#1}{#2} }
370 \NewExpandableDocumentCommand \TopMark { O{page} m }
371                     { \mark_use_top:nn {#1}{#2} }
```

(End of definition for `\TopMark` *,* `\FirstMark` *, and* `\LastMark`*. These functions are documented on page* *3.)*

`\IfMarksEqualTF`
We only provide a CamelCase command for the case with one region (optional) and one class. One could think of also providing a version for the general case with several optional arguments, but use cases for this are most likely rare, so not done yet.

```
372 \NewExpandableDocumentCommand \IfMarksEqualTF {O{page}mmm} {
373   \mark_if_eq:nnnnTF {#1}{#2}{#3}{#4}
374 }
```

(End of definition for `\IfMarksEqualTF`*. This function is documented on page* *4.)*

8 LaTeX 2_ε integration

8.1 Core LaTeX 2_ε integration

`\__mark_update_singlecol_structures:`
This command updates the mark structures if we are producing a single column document.

```
375 \cs_new_protected:Npn \__mark_update_singlecol_structures: {
```

First we update the page region (which also updates the previous-page.

The `\@outputbox` is normally in `\vbox` in LaTeX but we can't take that for granted (an amsmath test document changed it to an `\hbox` just to trip me up) so we are a little careful with unpack now.

```
376   \box_if_vertical:NTF \@outputbox
377       {
378         \__mark_update_structure_from_material:nn {page}
379           { \vbox_unpack:N  \@outputbox }
380       }
381       {
382         \__mark_update_structure_from_material:nn {page}
383           { \hbox_unpack:N  \@outputbox }
384       }
```

The we provide the necessary updates for the aliases.

```
385    \__mark_update_structure_alias:nn {previous-column}{previous-page}
386    \__mark_update_structure_alias:nn {column}{page}
387    \__mark_update_structure_alias:nn {first-column}{page}
388    \__mark_update_structure_alias:nn {last-column}{page}
389 ⟨*trace⟩
390 % move this into status itself?
391    \__mark_debug:n
392      {
393        \__mark_status:n
394          { in~ OR~ (
395            \legacy_if:nTF {@twoside}
396                { twoside-
397                  \int_if_odd:nTF \c@page
398                    { odd }{ even }
399                }
400                { oneside }
401              )
402          }
403      }
404 ⟨/trace⟩
405 }
```

(*End of definition for* __mark_update_singlecol_structures:.)

__mark_update_dblcol_structures: This commands handles the updates if we are doing two-column pages.

```
406 \cs_new_protected:Npn \__mark_update_dblcol_structures: {
```

First we update the `column` and `previous-column` regions using the material assembled in `\@outputbox`.

```
407    \box_if_vertical:NTF \@outputbox
408      {
409        \__mark_update_structure_from_material:nn {column}
410          { \vbox_unpack:N  \@outputbox }
411      }
412      {
413        \__mark_update_structure_from_material:nn {column}
414          { \hbox_unpack:N  \@outputbox }
415      }
```

How we have to update the alias regions depends on whether or not `\@opcol` was called to process the first column or to produce the completed page

```
416    \legacy_if:nTF {@firstcolumn}
417      {
```

If we are processing the first column then `column` is our `first-column` and there is no `last-column` yet, so we make those an error.

```
418        \__mark_update_structure_alias:nn {first-column}{column}
419        \__mark_update_structure_to_err:n {last-column}
420      }
421      {
```

If we produce the completed page then the `first-column` is the same as the new `previous-column`. However, the structure should already be correct if you think about

it (because is was set to `column` last time which is now the `previous-column`), thus there is no need to make an update.

```
422 %        \__mark_update_structure_alias:nn {first-column}{previous-column}
```

However, we now have a proper `last-column` so we assign that.

```
423        \__mark_update_structure_alias:nn {last-column}{column}
```

What now remains doing is to update the `page` and `previous-page` regions. For this we have to copy the settings in `page` into `previous-page` and then update `page` such that the top and first marks are taken from the `first-column` region and the last marks are taken from the `last-column` region. All this has to be done for all mark classes so we loop over our sequence.

Note that one loop is needed if we arrange the copy statements in a suitable way.

```
424        \seq_map_inline:Nn \g__mark_classes_seq
425          {
```

The `previous-page` updates need to come before the updates for `page` region because otherwise the values to copy are already overwritten. necessary values.

```
426          \tl_gset_eq:cc { g__mark_previous-page_top_    ##1 _tl }
427                         { g__mark_page_top_             ##1 _tl }
428          \tl_gset_eq:cc { g__mark_previous-page_first_  ##1 _tl }
429                         { g__mark_page_first_           ##1 _tl }
430          \tl_gset_eq:cc { g__mark_previous-page_last_   ##1 _tl }
431                         { g__mark_page_last_            ##1 _tl }
```

To update the `top` we only have to copy what is in `first-column`:

```
432          \tl_gset_eq:cc { g__mark_page_top_          ##1 _tl }
433                         { g__mark_first-column_top_  ##1 _tl }
434
```

Updating the `first` mark for the `page` region is more complicated. We first have to find out of there is any mark in the first column (this can be done by comparing the `top` and the `first` mark of of that region).

```
435          \tl_if_eq:ccTF { g__mark_first-column_top_    ##1 _tl }
436                         { g__mark_first-column_first_  ##1 _tl }
437            {
```

If there is no mark in the first column we copy the first mark of the last column. If that doesn't contain a mark we still get the right result because the first mark is then equal to the top mark.

```
438              \tl_gset_eq:cc { g__mark_page_first_         ##1 _tl }
439                             { g__mark_last-column_first_  ##1 _tl }
440            }
441            {
```

On the other hand, if there is a mark in the first column we copy over the `first` mark from that column.

```
442              \tl_gset_eq:cc { g__mark_page_first_          ##1 _tl }
443                             { g__mark_first-column_first_  ##1 _tl }
444            }
```

The logic for the `last` page mark is again simple, we can just copy the value in the `last` mark of the last column. If that column doesn't contain any marks, then the value in `last` will be automatically the same as the `last` from the first column.

```
445          \tl_gset_eq:cc { g__mark_page_last_          ##1 _tl }
446                         { g__mark_last-column_last_  ##1 _tl }
```

```
447               }
448            }
449  ⟨*trace⟩
450        \__mark_debug:n
451          {
452            \__mark_status:n
453              { in~ OR~ (
454                \legacy_if:nTF {@twoside}
455                     { twoside-
456                       \int_if_odd:nTF \c@page
457                          { odd }{ even }
458                     }
459                     { oneside }
460                     \space
461                     \legacy_if:nTF {@firstcolumn}
462                        { first~ }{ second~ }
463                        column )
464             }
465           }
466  ⟨/trace⟩
467  }
```

(*End of definition for* __mark_update_dblcol_structures:.)

```
468  ⟨@@=⟩
```

```
469  \cs_new_eq:NN  \@expl@@@mark@update@singlecol@structures@@
470                 \__mark_update_singlecol_structures:
```

(*End of definition for* \@expl@@@mark@update@singlecol@structures@@.)

```
471  \cs_new_eq:NN  \@expl@@@mark@update@dblcol@structures@@
472                 \__mark_update_dblcol_structures:
```

(*End of definition for* \@expl@@@mark@update@dblcol@structures@@.)

8.2   Other LaTeX 2ε output routines

This section will cover multicol and other packages altering or providing their own output routine. Not done yet.

8.3   Rollback information

```
473  ⟨latexrelease⟩\IncludeInRelease{0000/00/00}{ltmarks}%
474  ⟨latexrelease⟩                    {Undo~Marks~handling}
475  ⟨latexrelease⟩
```

We keep the interface commands around even if we roll back in case they are used in packages that don't roll back. Not likely to do a lot of good, but then there is not much we can do, but this at least they won't give unknown csname errors.

```
476  ⟨latexrelease⟩\DeclareRobustCommand \NewMarkClass[1]{}
477  ⟨latexrelease⟩\DeclareRobustCommand \InsertMark[2]{}
478  ⟨latexrelease⟩\RenewExpandableDocumentCommand \FirstMark { O{} m } { }
```

479 ⟨latexrelease⟩*\RenewExpandableDocumentCommand \LastMark { O{} m } { }*

480 ⟨latexrelease⟩*\RenewExpandableDocumentCommand \TopMark { O{} m } { }*

481 ⟨latexrelease⟩*\RenewExpandableDocumentCommand \IfMarksEqualTF { O{} mmm }{ }*

482 ⟨latexrelease⟩

Same here, this avoided extra roll back code in the OR.

483 ⟨latexrelease⟩*\let \@expl@@@mark@update@singlecol@structures@@ \relax*

484 ⟨latexrelease⟩*\let \@expl@@@mark@update@dblcol@structures@@ \relax*

485 ⟨latexrelease⟩

486 ⟨latexrelease⟩

487 ⟨latexrelease⟩*\EndModuleRelease*

488 \ExplSyntaxOff

489 ⟨/2ekernel | latexrelease⟩

Reset module prefix:

490 ⟨@@=⟩