A new implementation of LaTeX's tabular and array environment*

Frank Mittelbach David Carlisle†

Printed October 22, 2025

This file is maintained by the LaTeX Project team.
Bug reports can be opened (category `tools`) at
https://latex-project.org/bugs.html.

Abstract

This article describes an extended implementation of the LaTeX array– and tabular–environments. The special merits of this implementation are further options to format columns and the fact that fragile LaTeX–commands don't have to be `\protect`'ed any more within those environments.
The major part of the code for this package dates back to 1988—so does some of its documentation.

1 Introduction

This new implementation of the array– and tabular–environments is part of a larger project in which we are trying to improve the LaTeX-code in some aspects and to make LaTeX even easier to handle.

The reader should be familiar with the general structure of the environments mentioned above. Further information can be found in [3] and [1]. The additional options which can be used in the preamble as well as those which now have a slightly different meaning are described in table 1.

`\extrarowheight` Additionally we introduce a new parameter called `\extrarowheight`. If it takes a positive length, the value of the parameter is added to the normal height of every row of the table, while the depth will remain the same. This is important for tables with horizontal lines because those lines normally touch the capital letters. For example, we used `\setlength{\extrarowheight}{1pt}` in table 1.

We will discuss a few examples using the new preamble options before dealing with the implementation.

- If you want to use a special font (for example `\bfseries`) in a flushed left column, this can be done with `>{\bfseries}l`. You do not have to begin every entry of the column with `\bfseries` any more.

*This file has version number v2.6n, last revised 2025/09/25.

†David kindly agreed on the inclusion of the `\newcolumntype` implementation, formerly in `newarray.sty` into this package.

Unchanged options	
`l`	Left adjusted column.
`c`	Centered adjusted column.
`r`	Right adjusted column.
`p{width}`	Equivalent to `\parbox[t]{width}`.
`@{decl.}`	Suppresses inter-column space and inserts `decl.` instead.
New options	
`m{width}`	Defines a column of width `width`. Every entry will be centered in proportion to the rest of the line. It is somewhat like `\parbox{width}`.
`b{width}`	Coincides with `\parbox[b]{width}`.
`>{decl.}`	Can be used before an `l`, `r`, `c`, `p`, `m` or a `b` option. It inserts `decl.` directly in front of the entry of the column.
`<{decl.}`	Can be used after an `l`, `r`, `c`, `p{..}`, `m{..}` or a `b{..}` option. It inserts `decl.` right after the entry of the column.
`|`	Inserts a vertical line. The distance between two columns will be enlarged by the width of the line in contrast to the original definition of LaTeX.
`!{decl.}`	Can be used anywhere and corresponds with the `|` option. The difference is that `decl.` is inserted instead of a vertical line, so this option doesn't suppress the normally inserted space between columns in contrast to `@{...}`.
`w{align}{width}`	Sets the cell content in a box of the specified `width` aligned according to the `align` parameter which could be either `l`, `c` or `r`. Works essentially like `\makebox[width][align]{cell}` so silently overprints if the cell content is wider than the specified width. If that is not desired use `W` instead.
`W{align}{width}`	Like `w` but spits out an overfull box warning (and an overfullrule marker in draft mode) when the cell content is too wide to fit. This also means that the alignment is different if there is too much material, because it then always protrudes to the right!

Table 1: The preamble options.

- In columns which have been generated with `p`, `m` or `b`, the default value of `\parindent` is `0pt`. This can be changed with `>{\setlength{\parindent}{1cm}}p`.
- The `>`– and `<`–options were originally developed for the following application: `>{$}c<{$}` generates a column in math mode in a tabular–environment. If you use this type of a preamble in an array–environment, you get a column in LR mode because the additional $'s cancel the existing $'s.
- One can also think of more complex applications. A problem which has been mentioned several times in TeXhax can be solved with `>{\centerdots}c <{\endcenterdots}`. To center decimals at their decimal points you (only?) have

to define the following macros:

```
\catcode`\.\active\gdef.{\egroup\setbox2\hbox\bgroup}}
def\centerdots{\catcode`\.\active\setbox0\hbox\bgroup}
def\endcenterdots{\egroup\ifvoid2 \setbox2\hbox{0}\fi
  \ifdim \wd0>\wd2 \setbox2\hbox to\wd0{\unhbox2\hfill}\else
    \setbox0\hbox to\wd2{\hfill\unhbox0}\fi
  \catcode`\.12 \box0.\box2}
```

Warning: The code is bad, it doesn't work with more than one dot in a cell and doesn't work when the tabular is used in the argument of some other command. A much better version is provided in the `dcolumn.sty` by David Carlisle.

- Using `c!{\hspace{1cm}}c` you get space between two columns which is enlarged by one centimeter, while `c@{\hspace{1cm}}c` gives you exactly one centimeter space between two columns.
- A declaration like `w{l}{3cm}` (or even shorter `wl{3cm}`) works like an `l` column except that the width will always be `3cm` regardless of the cell content. Same with `w{c}` or `w{r}`. This means that it is easy to set up tables in which all columns have predefined widths.

1.1 A note on the allowed content of `>{...}` and `<{...}`

These specifiers are meant to hold declarations, such as `>{\itshape}`. They cannot end in commands that take arguments without providing these arguments as part of the `{...}`. It would be a mistaken assumption that they pick up all or parts of the alignment entry data if their argument is not provided. E.g., `>{\textbf}` would not make the whole column bold nor would it make the first character bold (technically it would try to bolden `\ignorespaces`). Thus, it would not fail with an error, but effectively the output would be wrong and not as expected.

1.2 The behavior of the \\ command

In the basic `tabular` implementation of LaTeX the `\\` command ending the rows of the `tabular` or `array` has a somewhat inconsistent behavior if its optional argument is used. The result then depends on the type of rightmost column and as remarked in Leslie Lamport's LaTeX manual [3] may not always produce the expected extra space.

Without the `array` package the extra space requested by the optional argument of `\\` is measured from the last baseline of the rightmost column (indicated by "x" in the following example). As a result, swapping the column will give different results:

```
\begin{tabular}[t]{lp{1cm}}
    1 & 1\newline x   \\[20pt]     2 & 2    \end{tabular}
\begin{tabular}[t]{p{1cm}l}
    1\newline 1 & x   \\[20pt]     2 & 2    \end{tabular}
```

If you run this without the array package you will get the following result:

1	1	1	x
	x	1	
		2	2
2	2		

In contrast, when the array package is loaded, the requested space in the optional argument is always measured from the baseline of the whole row and not from the last baseline of the rightmost column, thus swapping columns doesn't change the spacing and we same table height with an effective 8pt of extra space (as the second line already takes up 12pt of the requested 20pt):

1	1	1	x
	x	1	
2	2	2	2

This correction of behavior only makes a difference if the rightmost column is a `p`-column. Thus if you add the array package to an existing document, you should verify the spacing in all tables that have this kind of structure.

1.3 Defining new column specifiers

`\newcolumntype` Whilst it is handy to be able to type

```
>{⟨some declarations⟩}{c}<{⟨some more declarations⟩}
```

if you have a one-off column in a table, it is rather inconvenient if you often use columns of this form. The new version allows you to define a new column specifier, say `x`, which will expand to the primitives column specifiers.[1] Thus we may define

```
\newcolumntype{x}{>{⟨some declarations⟩}{c}<{⟨some more declarations⟩}}
```

One can then use the `x` column specifier in the preamble arguments of all `array` or `tabular` environments in which you want columns of this form.

It is common to need math-mode and LR-mode columns in the same alignment. If we define:

```
\newcolumntype{C}{>{$}c<{$}}
\newcolumntype{L}{>{$}l<{$}}
\newcolumntype{R}{>{$}r<{$}}
```

Then we can use `C` to get centered LR-mode in an `array`, or centered math-mode in a `tabular`.

The example given above for 'center decimal points' could be assigned to a `d` specifier with the following command.

```
\newcolumntype{d}{>{\centerdots}c<{\endcenterdots}}
```

[1] This command was named `\newcolumn` in the `newarray.sty`. At the moment `\newcolumn` is still supported (but gives a warning). In later releases it will vanish.

The above solution always centers the dot in the column. This does not look too good if the column consists of large numbers, but to only a few decimal places. An alternative definition of a `d` column is

```
\newcolumntype{d}[1]{>{\rightdots{#1}}r<{\endrightdots}}
```

where the appropriate macros in this case are:[2]

```
\def\coldot{.}% Or if you prefer, \def\coldot{\cdot}
{\catcode`\.=\active
  \gdef.{$\egroup\setbox2=\hbox to \dimen0 \bgroup$\coldot}}
\def\rightdots#1{%
  \setbox0=\hbox{$1$}\dimen0=#1\wd0
  \setbox0=\hbox{$\coldot$}\advance\dimen0 \wd0
  \setbox2=\hbox to \dimen0 {}%
  \setbox0=\hbox\bgroup\mathcode`\."8000 $}
\def\endrightdots{$\hfil\egroup\box0\box2}
```

Note that `\newcolumntype` takes the same optional argument as `\newcommand` which declares the number of arguments of the column specifier being defined. Now we can specify `d{2}` in our preamble for a column of figures to at most two decimal places.

A rather different use of the `\newcolumntype` system takes advantage of the fact that the replacement text in the `\newcolumntype` command may refer to more than one column. Suppose that a document contains a lot of `tabular` environments that require the same preamble, but you wish to experiment with different preambles. Lamport's original definition allowed you to do the following (although it was probably a mis-use of the system).

```
\newcommand{\X}{clr}
\begin{tabular}{\X} ...
```

`array.sty` takes great care **not** to expand the preamble, and so the above does not work with the new scheme. With the new version this functionality is returned:

```
\newcolumntype{X}{clr}
\begin{tabular}{X} ...
```

The replacement text in a `\newcolumntype` command may refer to any of the primitives of `array.sty` see table 1 on page 2, or to any new letters defined in other `\newcolumntype` commands.

`\showcols` A list of all the currently active `\newcolumntype` definitions is sent to the terminal and log file if the `\showcols` command is given.

1.4 Special variations of `\hline`

The family of `tabular` environments allows vertical positioning with respect to the baseline of the text in which the environment appears. By default the environment appears centered, but this can be changed to align with the first or last line in the environment by supplying a `t` or `b` value to the optional position argument. However, this does not work when the first or last element in the environment is a `\hline` command—in that case the environment is aligned at the horizontal rule.

[2] The package `dcolumn.sty` contains more robust macros based on these ideas.

Here is an example:

Tables with no hline commands used versus tables with some hline commands used.

```
Tables
\begin{tabular}[t]{l}
 with no\\ hline \\ commands \\ used
\end{tabular} versus tables
\begin{tabular}[t]{|l|}
 \hline
  with some \\ hline \\ commands \\
 \hline
\end{tabular} used.
```

`\firsthline` `\lasthline` Using `\firsthline` and `\lasthline` will cure the problem, and the tables will align properly as long as their first or last line does not contain extremely large objects.

Tables with no line commands used versus tables with some line commands used.

```
Tables
\begin{tabular}[t]{l}
  with no\\ line \\ commands \\ used
\end{tabular} versus tables
\begin{tabular}[t]{|l|}
 \firsthline
  with some \\ line   \\ commands \\
 \lasthline
\end{tabular} used.
```

`\extratabsurround` The implementation of these two commands contains an extra dimension, which is called `\extratabsurround`, to add some additional space at the top and the bottom of such an environment. This is useful if such tables are nested.

2 Final Comments

2.1 Handling of rules

There are two possible approaches to the handling of horizontal and vertical rules in tables:

1. rules can be placed into the available space without enlarging the table, or
2. rules can be placed between columns or rows thereby enlarging the table.

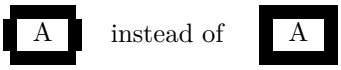
For vertical rules `array.sty` implements the second possibility while the default implementation in the LaTeX kernel implements the first concept. Both concepts have their merits but one has to be aware of the individual implications.

- With standard LaTeX adding vertical rules to a table will not affect the width of the table (unless double rules are used), e.g., changing a preamble from `lll` to `l|l|l` does not affect the document other than adding rules to the table. In contrast, with `array.sty` a table that just fit the `\textwidth` might now produce an overfull box.
- With standard LaTeX modifying the width of rules could result in ugly looking tables because without adjusting the `\tabcolsep`, etc. the space between rule and column could get too small (or too large). In fact even overprinting of text is possible. In contrast, with `array.sty` modifying any such length usually works well as the actual visual white space (from `\tabcolsep`, etc.) does not depend on the width of the rules.

- With standard LaTeX boxed tabulars actually have strange corners because the horizontal rules end in the middle of the vertical ones. This looks very unpleasant when a large `\arrayrulewidth` is chosen. In that case a simple table like

```
setlength{\arrayrulewidth}{5pt}
begin{tabular}{|l|}
 \hline  A \\  \hline
end{tabular}
```

  will produce something like



Horizontal rules produced with `\hline` add to the table height in both implementations but they differ in handling double `\hline`s. In contrast a `\cline` does not change the table height.[3]

2.2 Comparisons with older versions of `array.sty`

There are some differences in the way version 2.1 treats incorrect input, even if the source file does not appear to use any of the extra features of the new version.

- A preamble of the form `{wx*{0}{abc}yz}` was treated by versions prior to 2.1 as `{wx}`. Version 2.1 treats it as `{wxyz}`
- An incorrect positional argument such as `[Q]` was treated as `[c]` by `array.sty`, but is now treated as `[t]`.
- A preamble such as `{cc*{2}}` with an error in a $*$-form will generate different errors in the new version. In both cases the error message is not particularly helpful to the casual user.
- Repeated `<` or `>` constructions generated an error in earlier versions, but are now allowed in this package. `>{`⟨*decs1*⟩`}>{`⟨*decs2*⟩`}` is treated the same as `>{`⟨*decs2*⟩⟨*decs1*⟩`}`.
- The `\extracolsep` command does not work with the old versions of `array.sty`, see the comments in `array.bug`. With version 2.1 `\extracolsep` may again be used in `@`-expressions as in standard LaTeX, and also in `!`-expressions (but see the note below).

Prior to version 2.4f the space added by the optional argument to `\\` was added inside an m-cell if the last column was of type `m`. As a result that cell was vertically centered with that space inside, resulting in a strange offset. Since 2.4f, this space is now added after centering the cell.

A similar problem happened when `\extrarowheight` was used. For that reason m-cells now manually position the cell content which allows to ignore this extra space request during the vertical alignment.

[3] All a bit inconsistent, but nothing that can be changed after being 30+ years in existence.

2.3 Bugs and Features

- Error messages generated when parsing the column specification refer to the preamble argument **after** it has been re-written by the `\newcolumntype` system, not to the preamble entered by the user. This seems inevitable with any system based on pre-processing and so is classed as a **feature**.

- The treatment of multiple `<` or `>` declarations may seem strange at first. Earlier implementations treated `>{`⟨*decs1*⟩`}>{`⟨*decs2*⟩`}` the same as `>{`⟨*decs1*⟩⟨*decs2*⟩`}`. However this did not give the user the opportunity of overriding the settings of a `\newcolumntype` defined using these declarations. For example, suppose in an `array` environment we use a `C` column defined as above. The `C` specifies a centered text column, however `>{\bfseries}C`, which re-writes to `>{\bfseries}>{$}c<{$}` would not specify a bold column as might be expected, as the preamble would essentially expand to `\hfil$\bfseries$#$ $\hfil` and so the column entry would not be in the scope of the `\bfseries`! The present version switches the order of repeated declarations, and so the above example now produces a preamble of the form `\hfil$ $\bfseries#$ $\hfil`, and the dollars cancel each other out without limiting the scope of the `\bfseries`.

- The use of `\extracolsep` has been subject to the following two restrictions. There must be at most one `\extracolsep` command per `@`, or `!` expression and the command must be directly entered into the `@` expression, not as part of a macro definition. Thus `\newcommand{\ef}{\extracolsep{\fill}}` ... `@{\ef}` does not work with this package. However you can use something like `\newcolumntype{e}{@{\extracolsep{\fill}}` instead.

- As noted by the LaTeX book, for the purpose of `\multicolumn` each column with the exception of the first one consists of the entry and the *following* inter-column material. This means that in a tabular with the preamble `|l|l|l|l|` input such as `\multicolumn{2}{|c|}` in anything other than the first column is incorrect.

  In the standard array/tabular implementation this error is not so noticeable as that version contains negative spacing so that each `|` takes up no horizontal space. But since in this package the vertical lines take up their natural width one sees two lines if two are specified.

3 Support for tagged PDF

With version 2.6a the package is made tagging aware, which means that it will automatically produce tagged tables (necessary, for example, for accessibility) if tagging is requested via `\DocumentMetadata`.

More granular control, e.g., explicitly deciding which cells are header cells, etc., is currently under development, but syntax for this will not appear in this package. Instead it will become available across all tabular-generating packages and then automatically apply here as well.

Enabling LaTeX to automatically produce tagged PDF is a long-term project and this is a tiny step in this puzzle. For more information on the project and already available functionality, see https://latex-project.org/publications/indexbytopic/pdf and https://github.com/latex3/tagging-project.

4 The documentation driver file

The first bit of code contains the documentation driver file for TEX, i.e., the file that will produce the documentation you are currently reading. It will be extracted from this file by the `docstrip` program.

```
⟨*driver⟩
\NeedsTeXFormat{LaTeX2e}[2024/06/01]
```

We switched from ltxdoc to l3doc to get support for code written in the L3 programming layer. The first is that we are currently missing `\MaintainedByLaTeXTeam`, so we have to provide that for now.

```
\documentclass{l3doc}

% currently missing in l3doc
\makeatletter
\def\MaintainedBy#1{\gdef\@maintainedby{#1}}
\let\@maintainedby\@empty
\def\MaintainedByLaTeXTeam#1{%
{\gdef\@maintainedby{%
This file is maintained by the \LaTeX{} Project team.\\%
Bug reports can be opened (category \texttt{#1}) at\\%
\url{https://latex-project.org/bugs.html}.}}}
\def\@maketitle{%
  \newpage
  \null
  \vskip 2em%
  \begin{center}%
  \let \footnote \thanks
    {\LARGE \@title \par}%
    \vskip 1.5em%
    {\large
      \lineskip .5em%
      \begin{tabular}[t]{c}%
        \@author
      \end{tabular}\par}%
    \vskip 1em%
    {\large \@date}%
    \ifx\@maintainedby\@empty
    \else
    \vskip 1em%
    \fbox{\fbox{\begin{tabular}{@{}l@{}}\@maintainedby\end{tabular}}}%
    \fi
  \end{center}%
  \par
  \vskip 1.5em}
\makeatother

% undo the default is not used:

\IfFormatAtLeastTF {2020/10/01}
 {\AtBeginDocument[ltxdoc]{\DeleteShortVerb{\|}} }
 {\AtBeginDocument{\DeleteShortVerb{\|}} }

 \usepackage{array}
```

```
 % Allow large table at bottom
 \renewcommand{\bottomfraction}{0.7}

\EnableCrossrefs
 %\DisableCrossrefs   % Say \DisableCrossrefs if index is ready

\RecordChanges                  % Gather update information

\CodelineIndex                  % Index code by line number

 %\OnlyDescription    % comment out for implementation details
 %\OldMakeindex       % use if your MakeIndex is pre-v2.9

\begin{document}
   \DocInput{array.dtx}
\end{document}
⟨/driver⟩
```

5 A note on the updates done December 2023

We introduced support for tagged PDF and at the same time we added code to determine row and column numbers for each cell in preparation for supporting formatting or type specifications for individual cells (or group of cells) from the outside, e.g., "rows 1, 2, and 10 are header rows" (syntax to be decided).

This new code is already written with L3 programming layer conventions while most of the legacy code is still as it was before. This make the code currently somewhat clattered, unfortunately. Eventually this will all move to L3 programming layer but this will take time.

```
⟨@@=tbl⟩
\ExplSyntaxOn
```

6 The construction of the preamble

It is obvious that those environments will consist mainly of an `\halign`, because TEX typesets tables using this primitive. That is why we will now take a look at the algorithm which determines a preamble for a `\halign` starting with a given user preamble using the options mentioned above.

The current version is defined at the top of the file looking something like this

```
⟨*package⟩
%\NeedsTeXFormat{LaTeX2e}[1994/05/13]
%\ProvidesPackage{array}[\filedate\space version\fileversion]
```

The most interesting macros of this implementation are without doubt those which are responsible for the construction of the preamble for the `\halign`. The underlying algorithm was developed by LAMPORT (resp. KNUTH, see texhax V87#??), and it has been extended and improved.

The user preamble will be read token by token. A token is a single character like `c` or a block enclosed in `{...}`. For example the preamble of `\begin{tabular}{lc||c@{\hspace{1cm}}}` consists of the token `l`, `c`, `|`, `|`, `@` and `\hspace{1cm}`.

The currently used token and the one, used before, are needed to decide on how the construction of the preamble has to be continued. In the example mentioned above the `l` causes the preamble to begin with `\hskip\tabcolsep`. Furthermore `# \hfil` would be appended to define a flush left column. The next token is a `c`. Because it was preceded by an `l` it generates a new column. This is done with `\hskip \tabcolsep & \hskip \tabcolsep`. The column which is to be centered will be appended with `\hfil # \hfil`. The token `|` would then add a space of `\hskip \tabcolsep` and a vertical line because the last tokens was a `c`. The following token `|` would only add a space `\hskip \doublerulesep` because it was preceded by the token `|`. We will not discuss our example further but rather take a look at the general case of constructing preambles.

The example shows that the desired preamble for the `\halign` can be constructed as soon as the action of all combinations of the preamble tokens are specified. There are 18 such tokens so we have $19 \cdot 18 = 342$ combinations if we count the beginning of the preamble as a special token. Fortunately, there are many combinations which generate the same spaces, so we can define token classes. We will identify a token within a class with a number, so we can insert the formatting (for example of a column). Table 2 lists all token classes and their corresponding numbers.

token	\@chclass	\@chnum	token	\@chclass	\@chnum
c	0	0	Start	4	—
l	0	1	@-arg	5	—
r	0	2	!	6	—
m-arg	0	3	@	7	—
p-arg	0	4	<	8	—
b-arg	0	5	>	9	—
|	1	0	m	10	3
!-arg	1	1	p	10	4
<-arg	2	—	b	10	5
>-arg	3	—			

Table 2: Classes of preamble tokens

\@chclass
\@chnum
\@lastchclass

The class and the number of the current token are saved in the count registers `\@chclass` and `\@chnum`, while the class of the previous token is stored in the count register `\@lastchclass`. All of the mentioned registers are already allocated in the LaTeX format, which is the reason why the following three lines of code are commented out. Later throughout the text I will not mention it again explicitly whenever I use a % sign. These parts are already defined in the LaTeX format.

```
% \newcount \@chclass
% \newcount \@chnum
% \newcount \@lastchclass
```

(*End of definition for* `\@chclass`, `\@chnum`, *and* `\@lastchclass`.)

\@addtopreamble

We will save the already constructed preamble for the `\halign` in the global macro `\@preamble`. This will then be enlarged with the command `\@addtopreamble`.

```
\def\@addtopreamble#1{\xdef\@preamble{\@preamble #1}}
```

(*End of definition for* `\@addtopreamble`.)

6.1 The character class of a token

\@testpach

With the help of \@lastchclass we can now define a macro which determines the class and the number of a given preamble token and assigns them to the registers \@chclass and \@chnum.

```
\ExplSyntaxOff
\def\@testpach{\@chclass
```

First we deal with the cases in which the token (#1) is the argument of !, @, < or >. We can see this from the value of \@lastchclass:

```
 \ifnum \@lastchclass=6 \@ne \@chnum \@ne \else
  \ifnum \@lastchclass=7 5 \else
   \ifnum \@lastchclass=8 \tw@ \else
    \ifnum \@lastchclass=9 \thr@@
```

Otherwise we will assume that the token belongs to the class 0 and assign the corresponding number to \@chnum if our assumption is correct.

```
    \else \z@
```

If the last token was a p, m or a b, \@chnum already has the right value. This is the reason for the somewhat curious choice of the token numbers in class 10.

```
    \ifnum \@lastchclass = 10 \else
```

Otherwise we will check if \@nextchar is either a c, l or an r. Some applications change the catcodes of certain characters like "@" in amstex.sty. As a result the tests below would fail since they assume non-active character tokens. Therefore we evaluate \@nextchar once thereby turning the first token of its replacement text into a char. At this point here this should have been the only char present in \@nextchar which put into via a \def.

```
    \edef\@nextchar{\expandafter\string\@nextchar}%
    \@chnum
    \if \@nextchar c\z@ \else
     \if \@nextchar l\@ne \else
      \if \@nextchar r\tw@ \else
```

If it is a different token, we know that the class was not 0. We assign the value 0 to \@chnum because this value is needed for the |–token. Now we must check the remaining classes. Note that the value of \@chnum is insignificant here for most classes.

```
    \z@ \@chclass
    \if\@nextchar |\@ne \else
     \if \@nextchar !6 \else
      \if \@nextchar @7 \else
       \if \@nextchar <8 \else
        \if \@nextchar >9 \else
```

The remaining permitted tokens are p, m and b (class 10).

```
   10
   \@chnum
   \if \@nextchar m\thr@@ \else
    \if \@nextchar p4 \else
     \if \@nextchar b5 \else
```

Now the only remaining possibility is a forbidden token, so we choose class 0 and number 0 and give an error message. Then we finish the macro by closing all \if's.

```
    \z@ \@chclass \z@ \@preamerr \z@ \fi \fi \fi \fi
    \fi \fi  \fi  \fi  \fi  \fi  \fi \fi \fi \fi \fi \fi}
```

```
\ExplSyntaxOn
```

(*End of definition for* `\@testpach`.)

6.2 Multiple columns (∗–form)

`\@xexpast` `\the@toks` `\the@toksz`

Now we discuss the macro that deletes all forms of type `*{`*N*`}{`*String*`}` from a user preamble and replaces them with *N* copies of *String*. Nested ∗–expressions are dealt with correctly, that means ∗–expressions are not substituted if they are in explicit braces, as in `@{*}`.

This macro is called via `\@xexpast`⟨*preamble*⟩`*0x\@@@@`. The ∗–expression `*0x` is being used to terminate the recursion, as we shall see later, and `\@@@@` serves as an argument delimiter. `\@xexpast` has four arguments. The first one is the part of the user preamble before the first ∗–expression while the second and third ones are the arguments of the first ∗–expression (that is *N* and *String* in the notation mentioned above). The fourth argument is the rest of the preamble.

```
\def\@xexpast#1*#2#3#4\@@{%
```

The number of copies of *String* (`#2`) that are to be produced will be saved in a count register.

```
  \@tempcnta #2
```

We save the part of the preamble which does not contain a ∗–form (`#1`) in a Plain TeX token register. We also save *String* (`#3`) using a LaTeX token register.

```
  \toks@={#1}\@temptokena={#3}%
```

Now we have to use a little trick to produce *N* copies of *String*. We could try `\def\@tempa{#1}` and then *N* times `\edef\@tempa{\@tempa#3}`. This would have the undesired effect that all macros within `#1` and `#3` would be expanded, although, for example, constructions like `@{..}` are not supposed to be changed. That is why we `\let` two control sequences to be equivalent to `\relax`.

```
  \let\the@toksz\relax \let\the@toks\relax
```

Then we ensure that `\@tempa` contains `{\the@toksz\the@toks...\the@toks}` (the macro `\the@toks` exactly *N* times) as substitution text.

```
  \def\@tempa{\the@toksz}%
  \ifnum\@tempcnta >0 \@whilenum\@tempcnta >0\do
    {\edef\@tempa{\@tempa\the@toks}\advance \@tempcnta \m@ne}%
```

If *N* was greater than zero we prepare for another call of `\@xexpast`. Otherwise we assume we have reached the end of the user preamble, because we had appended `*0x\@@@@` when we first called `\@xexpast`. In other words: if the user inserts `*{0}{..}` in his preamble, LaTeX ignores the rest of it.

```
      \let \@tempb \@xexpast \else
      \let \@tempb \@xexnoop \fi
```

Now we will make sure that the part of the user preamble, which was already dealt with, will be saved again in `\@tempa`.

```
  \def\the@toksz{\the\toks@}\def\the@toks{\the\@temptokena}%
  \edef\@tempa{\@tempa}%
```

We have now evaluated the first ∗–expression, and the user preamble up to this point is saved in `\@tempa`. We will put the contents of `\@tempa` and the rest of the user preamble together and work on the result with `\@tempb`. This macro either corresponds to `\@xexpast`, so that the next ∗–expression is handled, or to the macro `\@xexnoop`, which only ends the recursion by deleting its argument.

```
111    \expandafter \@tempb \@tempa #4\@@}
```

(*End of definition for* `\@xexpast`, `\the@toks`, *and* `\the@toksz`.)

`\@xexnoop` So the first big problem is solved. Now it is easy to specify `\@xexnoop`. Its argument is delimited by `\@@@@` and it simply expands to nothing.

```
112 %  \def\@xexnoop#1\@@{}
```

(*End of definition for* `\@xexnoop`.)

7 The insertion of declarations (>, <, !, @)

The preamble will be enlarged with the help of `\xdef`, but the arguments of `>`, `<`, `!` and `@` are not supposed to be expanded during the construction (we want an implementation that doesn't need a `\protect`). So we have to find a way to inhibit the expansion of those arguments.

We will solve this problem with token registers. We need one register for every `!` and `@`, while we need two for every `c`, `l`, `r`, `m`, `p` or `b`. This limits the number of columns of a table because there are only 256 token registers. But then, who needs tables with more than 100 columns?

One could also find a solution which only needs two or three token registers by proceeding similarly as in the macro `\@xexpast` (see page 13). The advantage of our approach is the fact that we avoid some of the problems that arise with the other method[4].

So how do we proceed? Let us assume that we had `!{foo}` in the user preamble and say we saved `foo` in token register 5. Then we call `\@addtopreamble{\the@toks5}` where `\the@toks` is defined in a way that it does not expand (for example it could be equivalent to `\relax`). Every following call of `\@addtopreamble` leaves `\the@toks5` unchanged in `\@preamble`. If the construction of the preamble is completed we change the definition of `\the@toks` to `\the\toks` and expand `\@preamble` for the last time. During this process all parts of the form `\the@toks`⟨*Number*⟩ will be substituted by the contents of the respective token registers.

As we can see from this informal discussion the construction of the preamble has to take place within a group, so that the token registers we use will be freed later on. For that reason we keep all assignments to `\@preamble` global; therefore the replacement text of this macro will remain the same after we leave the group.

`\count@` We further need a count register to remember which token register is to be used next. This will be initialized with −1 if we want to begin with the token register 0. We use the Plain TEX scratch register `\count@` because everything takes place locally. All we have to do is insert `\the@toks \the \count@` into the preamble. `\the@toks` will remain unchanged and `\the\count@` expands into the saved number.

(*End of definition for* `\count@`.)

[4] Maybe there are also historical reasons.

\prepnext@tok

The macro \prepnext@tok is in charge of preparing the next token register. For that purpose we increase \count@ by 1:

```
\def\prepnext@tok{\advance \count@ \@ne
```

Then we locally delete any contents the token register might have.

```
  \toks\count@{}}
```

(*End of definition for* \prepnext@tok.)

\save@decl

During the construction of the preamble the current token is always saved in the macro \@nextchar (see the definition of \@mkpream on page 18). The macro \save@decl saves it into the next free token register, i.e. in \toks\count@.

```
\def\save@decl{\toks\count@ \expandafter{\@nextchar}}
```

The reason for the use of \relax is the following hypothetical situation in the preamble: ..\the\toks1\the\toks2.. TEX expands \the\toks2 first in order to find out if the digit 1 is followed by other digits. E.g. a 5 saved in the token register 2 would lead TEX to insert the contents of token register 15 instead of 1 later on.

The example above referred to an older version of \save@decl which inserted a \relax inside the token register. This is now moved to the places where the actual token registers are inserted (look for \the@toks) because the old version would still make @ expressions to moving arguments since after expanding the second register while looking for the end of the number the contents of the token register is added so that later on the whole register will be expanded. This serious bug was found after nearly two years international use of this package by Johannes Braams.

(*End of definition for* \save@decl.)

How does the situation look like, if we want to add another column to the preamble, i.e. if we have found a c, l, r, p, m or b in the user preamble? In this case we have the problem of the token register from >{..} and <{..} having to be inserted at this moment because formatting instructions like \hfil have to be set around them. On the other hand it is not known yet, if any <{..} instruction will appear in the user preamble at all.

We solve this problem by adding two token registers at a time. This explains, why we have freed the token registers in \prepnext@tok.

\insert@column
\@sharp
\textonly@unskip
\@protected@firstofone

We now define the macro \insert@column which will do this work for us.

```
\def\insert@column{%
```

For tagging we insert as special socket, that adds the necessary PDF tag at the beginning of the cell if tagging is enabled.

```
  \UseTaggingSocket{tbl/cell/begin}%
```

Next we have to insert the toks register holding the content of >{...}. Here, we assume that the count register \@tempcnta has saved the value \count@ − 1.

To keep TEX happy if there is a look ahead in the tabular preamble, i.e., starting in >{...}, which uses the Appendix D trick (for example, anything with a trailing optional argument defined by ltcmd), we wrap everything here in a protected version of \@firstofone. TEX otherwise can get confused about the value of the master counter, and we get some strange errors. We suspected that there was an underlying issue is the TEX engine, but it turned out to be rather hard to get to the bottom of it, because the master counter is not accessible through TEX's tracing tools. Thus, all we could do was producing various example documents, observing results, as well as staring at a printout of the TEX program. As an example, without this approach, something like

```
\NewDocumentCommand\foo{o}{x}
\begin{tabular}{>{\foo}l}
   Foo
\end{tabular}
```

failed. That can be fixed by adding a `\relax` after the `\@tempcnta`, but that then leads to issues if you are collecting whole cells (tagging code or collcell), where you can no longer alter the meaning of `\cr` as the master counter goes wrong due to an obscure bug (or perhaps, say, an undocumented feature of TEX). Eventually, we were able to pin down the root cause and really understand why `\@protected@firstofone` solves the problem, even though it looks like a nonsense addition to the code that does nothing useful.[5]

The problem is that TEX tries to conserve stack space, and when the last token of an existing token list is a macro, then this token list is *first* removed from memory (reducing the stack) *before* the macro replacement text (as a new token list) is given to the parser adding a new stack level. This is done using the routine `end_token_list` in the TEX program and ending the u-part of an `\halign` column with this routine immediately sets the *master counter* used by alignments to zero (see chapter 22 and Appendix D of the TEXbook). This means that technically the expansion of the last token in the u-part (if it is a macro) is not executed in the context of the u-part, but in the context of the alignment entry in the document. That normally doesn't make any difference whatsoever — unless you play around (as we sometimes have to) with tricks like those from Appendix D.

To illustrate the issue we show a bit of strange low-level plain TEX code.[6] Below are two very special grouping commands that are like `\bgroup` and `\egroup` but also affect the alignment master counter when expanded (see TEXbook p.385). If one of them is used as the last macro in the u-part of a column, then you get strange errors that you shouldn't get.

```
\def\bbgroup{{\ifnum0=`}\fi}
\def\eegroup{\ifnum0=`{\fi}}

% Fails with an error message, but there should be none:
\halign{%
  \message{u-part^^J}%
  \bbgroup              % <-- in the u-part
  \eegroup              % <-- in the u-part
  #%
  \message{v-part^^J}%
  \hfill\cr
  \message{body^^J}x
  \cr
}

% Fails but should work, the v-part is never reached:
```

[5] So it is a TEX engine bug that was in there from day one, or if you like, it is a hidden feature that is not explained; neither in the TEXbook nor in the program code. We don't really expect this to change in TEX after such a long time, other than perhaps documenting it as a feature, so this is a proper solution to the problem and not just a workaround.

[6] If all of this looks mighty strange to you, don't worry. You will be unlikely to need to know about it. It is just there so that programmers at some point in the future do not have to wonder too much why there is this odd `\@protected@firstofone` that apparently does nothing useful. It took us several nights of head scratching to come up with these minimal examples and then some more time to understand what the heck is going on inside TEX—thanks to Bruno for the right ideas on the latter.

```
\halign{%
  \message{u-part^^J}%
  \bbgroup              % <-- in the u-part
  #%
  \message{v-part^^JJ}%
  \eegroup              % <-- in the v-part
  \hfill\cr
  \message{body^^J}x
  \cr
}
```

So the trick we use now is making `\@protected@firstofone` the last macro in the u-part, i.e., before the `\@sharp`. That way its argument is always fully expanded as part of the alignment entry and not as part of the u-part and this way we know exactly what the master counter value is at this point, regardless of the content of `>{...}`.

```
\@protected@firstofone { \the@toks \the \@tempcnta \ignorespaces }
```

Next follows the `#` sign which specifies the place where the text of the column shall be inserted. To avoid errors during the expansions in `\@addtopreamble` we hide this sign in the command `\@sharp` which is temporarily occupied with `\relax` during the build-up of the preamble. To remove unwanted spaces before and after the column in text mode, we set an `\ignorespaces` in front (see above) and a `\unskip` afterwards; in math mode, the latter is suppressed while the `\ignorespaces` makes no difference.

```
\@sharp \textonly@unskip
```

Then the second `token` register follows whose number should be saved in `\count@`. We make sure that there will be no further expansion after reading the number, by finishing with `\relax`. The case above is not critical since it is ended by `\ignorespaces`.

```
\the@toks \the \count@ \relax
```

And another socket for tagging that adds the necessary closing tag if enabled.

```
\UseTaggingSocket{tbl/cell/end}%
}
```

Do the unskip only if we are in hmode:

```
\protected\def\textonly@unskip{\ifhmode\unskip\fi}
```

We need an engine-protected function that is just `\@firstofone`:

```
\protected\long\def\@protected@firstofone#1{#1}
```

(*End of definition for* `\insert@column` *and others.*)

`\insert@pcolumn` Handling pcolumn-cells needs slightly different handling when doing tagging. Rather than changing the plugs in `\insert@column` back and forth, we simply use a different version of `\insert@column` that has its own sockets.

```
\def\insert@pcolumn{%
  \UseTaggingSocket{tbl/pcell/begin}%
  \the@toks \the \@tempcnta \relax
  \ignorespaces \@sharp \unskip
  \the@toks \the \count@ \relax
  \UseTaggingSocket{tbl/pcell/end}%
}
```

(*End of definition for* `\insert@pcolumn`.)

7.1 The separation of columns

\@addamp In the preamble a & has to be inserted between any two columns; before the first column there should not be a &. As the user preamble may start with a | we have to remember somehow if we have already inserted a # (i.e. a column). This is done with the boolean variable \if@firstamp that we test in \@addamp, the macro that inserts the &.

```
%    \newif \@iffirstamp
\def\@addamp {
  \if@firstamp
    \@firstampfalse
  \else
```

If we are after the first column we have to insert a & and also update the cell data.

```
    \edef\@preamble{\@preamble &
      \noexpand\tbl_update_cell_data: }
  \fi
}
```

(*End of definition for* \@addamp.)

\@acol \@acolampacol \col@sep We will now define some abbreviations for the extensions, appearing most often in the preamble build-up. Here \col@sep is a dimen register which is set equivalent to \arraycolsep in an array–environment, otherwise it is set equivalent to \tabcolsep.

```
\newdimen\col@sep
\def\@acol{\@addtopreamble{\hskip\col@sep}}
%    \def\@acolampacol{\@acol\@addamp\@acol}
```

(*End of definition for* \@acol , \@acolampacol , *and* \col@sep.)

7.2 The macro \@mkpream

\@mkpream *The code below has been replaced long time ago by an extended version further down but the code and its documentation was left here for reference. It is now commented out to avoid confusion.*

Now we can define the macro which builds up the preamble for the \halign. First we \the@toks initialize \@preamble, \@lastchclass and the boolean variable \if@firstamp.

```
%\def\@mkpream#1{\gdef\@preamble{}\@lastchclass 4 \@firstamptrue
```

During the build-up of the preamble we cannot directly use the # sign; this would lead to an error message in the next \@addtopreamble call. Instead, we use the command \@sharp at places where later a # will be. This command is at first given the meaning \relax; therefore it will not be expanded when the preamble is extended. In the macro \@array, shortly before the \halign is carried out, \@sharp is given its final meaning.

In a similar way, we deal with the commands \@startpbox and \@endpbox, although the reason is different here: these macros expand in many tokens which would delay the build-up of the preamble.

```
%   \let\@sharp\relax\let\@startpbox\relax\let\@endpbox\relax
```

Two more are needed to deal with the code that handles struts for extra space after a row from \\[<space>] (\do@row@strut) and code that manages m-cells depending on their heights (\ar@align@mcell).

```
%   \let\do@row@strut\relax
%   \let\ar@align@mcell\relax
```

Now we remove possible ∗-forms in the user preamble with the command `\@xexpast`. As we already know, this command saves its result in the macro `\@tempa`.

```
%   \@xexpast #1*0x\@@
```

Afterwards we initialize all registers and macros, that we need for the build-up of the preamble. Since we want to start with the token register 0, `\count@` has to contain the value −1.

```
%   \count@\m@ne
%   \let\the@toks\relax
```

Then we call up `\prepnext@tok` in order to prepare the token register 0 for use.

```
%   \prepnext@tok
```

To evaluate the user preamble (without stars) saved in `\@tempa` we use the LaTeX–macro `\@tfor`. The strange appearing construction with `\expandafter` is based on the fact that we have to put the replacement text of `\@tempa` and not the macro `\@tempa` to this LaTeX–macro.

```
%   \expandafter \@tfor \expandafter \@nextchar
%    \expandafter :\expandafter =\@tempa \do
```

The body of this loop (the group after the `\do`) is executed for one token at a time, whereas the current token is saved in `\@nextchar`. At first we evaluate the current token with the already defined macro `\@testpach`, i.e. we assign to `\@chclass` the character class and to `\@chnum` the character number of this token.

```
%   {\@testpach
```

Then we branch out depending on the value of `\@chclass` into different macros that extend the preamble respectively.

```
%   \ifcase \@chclass \@classz \or \@classi \or \@classii
%     \or \save@decl \or \or \@classv \or \@classvi
%     \or \@classvii \or \@classviii  \or \@classix
%     \or \@classx \fi
```

Two cases deserve our special attention: Since the current token cannot have the character class 4 (start) we have skipped this possibility. If the character class is 3, only the content of `\@nextchar` has to be saved into the current token register; therefore we call up `\save@decl` directly and save a macro name. After the preamble has been extended we assign the value of `\@chclass` to the counter `\@lastchclass` to assure that this information will be available during the next run of the loop.

```
%   \@lastchclass\@chclass}%
```

After the loop has been finished space must still be added to the created preamble, depending on the last token. Depending on the value of `\@lastchclass` we perform the necessary operations.

```
%   \ifcase\@lastchclass
```

If the last class equals 0 we add a `\hskip \col@sep`.

```
%   \@acol \or
```

If it equals 1 we do not add any additional space so that the horizontal lines do not exceed the vertical ones.

```
%   \or
```

Class 2 is treated like class 0 because a `<{...}` can only directly follow after class 0.

```
%   \@acol \or
```

Most of the other possibilities can only appear if the user preamble was defective. Class 3 is not allowed since after a `>{..}` there must always follow a `c`, `l`, `r`, `p`,`m` or `b`. We report an error and ignore the declaration given by `{..}`.

```
%   \@preamerr \thr@@ \or
```

If `\@lastchclass` is 4 the user preamble has been empty. To continue, we insert a `#` in the preamble.

```
%   \@preamerr \tw@ \@addtopreamble\@sharp \or
```

Class 5 is allowed again. In this case (the user preamble ends with `@{..}`) we need not do anything.

```
%   \or
```

Any other case means that the arguments to `@`, `!`, `<`, `>`, `p`, `m` or `b` have been forgotten. So we report an error and ignore the last token.

```
%   \else  \@preamerr \@ne \fi
```

Now that the build-up of the preamble is almost finished we can insert the token registers and therefore redefine `\the@toks`. The actual insertion, though, is performed later.

```
%   \def\the@toks{\the\toks}}
```

(*End of definition for* `\@mkpream` *and* `\the@toks`.)

8 The macros \@classz to \@classx

The preamble is extended by the macros `\@classz` to `\@classx` which are called by `\@mkpream` depending on `\@lastchclass` (i.e. the character class of the last token).

`\@classx` First we define `\@classx` because of its important rôle. When it is called we find that the current token is `p`, `m` or `b`. That means that a new column has to start.

```
\def\@classx{%
```

Depending on the value of `\@lastchclass` different actions must take place:

```
  \ifcase \@lastchclass
```

If the last character class was 0 we separate the columns by `\hskip\col@sep` followed by `&` and another `\hskip\col@sep`.

```
  \@acolampacol \or
```

If the last class was class 1 — that means that a vertical line was drawn, — before this line a `\hskip\col@sep` was inserted. Therefore there has to be only a `&` followed by `\hskip\col@sep`. But this `&` may be inserted only if this is not the first column. This process is controlled by `\if@firstamp` in the macro `\addamp`.

```
  \@addamp \@acol \or
```

Class 2 is treated like class 0 because `<{...}` can only follow after class 0.

```
  \@acolampacol \or
```

Class 3 requires no actions because all things necessary have been done by the preamble token `>`.

```
  \or
```

Class 4 means that we are at the beginning of the preamble. Therefore we start the preamble with `\hskip\col@sep` and then call `\@firstampfalse`. This makes sure that a later `\@addamp` inserts the character `&` into the preamble.

```
  \@acol \@firstampfalse \or
```

For class 5 tokens only the character & is inserted as a column separator. Therefore we call \@addamp.

```
\@addamp
```

Other cases are impossible. For an example \@lastchclass = 6—as it might appear in a preamble of the form ...!p...—p would have been taken as an argument of ! by \@testpach.

```
\fi}
```

(*End of definition for* \@classx.)

Until the sockets are in the release we test for existence and declare them.

```
\str_if_exist:cF { l__socket_tagsupport/math/luamml/array/save_plug_str }
 {
   \NewSocket{tagsupport/math/luamml/array/save}{0}
   \NewSocket{tagsupport/math/luamml/array/finalize}{0}
   \NewSocket{tagsupport/math/luamml/array/initcol}{0}
   \NewSocket{tagsupport/math/luamml/array/finalizecol}{1}
   \AssignSocketPlug{tagsupport/math/luamml/array/finalizecol}{noop}
 }
```

\@classz If the character class of the last token is 0 we have c, l, r or an argument of m, b or p. In the first three cases the preamble must be extended the same way as if we had class 10. The remaining two cases do not require any action because the space needed was generated by the last token (i.e. m, b or p). Since \@lastchclass has the value 10 at this point nothing happens when \@classx is called. So the macro \@chlassz may start like this:

```
\def\@classz{\@classx
```

According to the definition of \insert@column we must store the number of the token register in which a preceding >{..} might have stored its argument into \@tempcnta.

```
   \@tempcnta \count@
```

To have \count@ = \@tmpcnta + 1 we prepare the next token register.

```
   \prepnext@tok
```

Now the preamble must be extended with the column whose format can be determined by \@chnum.

```
   \@addtopreamble{\ifcase \@chnum
```

If \@chnum has the value 0 a centered column has to be generated. So we begin with stretchable space.

```
       \hfil
```

We also add a space of 1sp just in case the first thing in the cell is a command doing an \unskip.

```
       \hskip1sp%
```

The command \d@llarbegin follows expanding into \begingroup (in the tabular–environment) or into $. Doing this (provided an appropriate setting of \d@llarbegin) we achieve that the contents of the columns of an array–environment are set in math mode while those of a tabular–environment are set in LR mode.

```
       \d@llarbegin
```

Now we insert the contents of the two token registers and the symbol for the column entry (i.e. # or more precise \@sharp) using \insert@column.

```
       \insert@column
```

We end this case with `\d@llarend` and `\hfil` where `\d@llarend` again is either `$` or `\endgroup`. The strut to enforce a regular row height is placed between the two.

```
194         \d@llarend
195         \do@row@strut \hfil \or
```

The templates for `l` and `r` (i.e. `\@chnum` 1 or 2) are generated the same way. Since one `\hfil` is missing the text is moved to the relevant side. The `\kern\z@` is needed in case of an empty column entry. Otherwise the `\unskip` in `\insert@column` removes the `\hfil`. Changed to `\hskip1sp` so that it interacts better with `\@bsphack`.

```
196         \hskip1sp\d@llarbegin\insert@column\d@llarend
197         \do@row@strut \hfil \or
198         \hfil\hskip1sp\d@llarbegin\insert@column\d@llarend
199         \do@row@strut \or
```

The templates for `p`, `m` and `b` mainly consist of a box. In case of `m` it is generated by `\vcenter`. This command is allowed only in math mode. Therefore we start with a `$`.

```
200       \setbox\ar@mcellbox\vbox
```

The part of the templates which is the same in all three cases (`p`, `m` and `b`) is built by the macros `\@startpbox` and `\@endpbox`. `\@startpbox` has an argument: the width of the column which is stored in the current token (i.e. `\@nextchar`). Between these two macros we find the well known `\insert@column` or rather the variant for tagging: `\insert@pcolumn`. The strut is placed after the box.

```
201       \@startpbox{\@nextchar}\insert@pcolumn \@endpbox
202       \ar@align@mcell
203       \do@row@strut \or
```

The templates for `p` and `b` are generated in the same way though we do not need the `$` characters because we use `\vtop` or `\vbox`.

```
204     \vtop \@startpbox{\@nextchar}\insert@pcolumn \@endpbox\do@row@strut \or
205     \vbox \@startpbox{\@nextchar}\insert@pcolumn \@endpbox\do@row@strut
```

Other values for `\@chnum` are impossible. Therefore we end the arguments to `\@addtopreamble` and `\ifcase`. Before we come to the end of `\@classz` we have to prepare the next token register.

```
206   \fi}\prepnext@tok}
```

(*End of definition for* `\@classz`.)

`\ar@mcellbox` When dealing with m-cells we need a box to measure the cell height.

```
207 \newbox\ar@mcellbox
```

(*End of definition for* `\ar@mcellbox`.)

`\ar@align@mcell` M-cells are supposed to be vertically centered within the table row. In the original implementation that was done using `\vcenter` but the issue with that approach is that it centers the material based on the math-axis. In most situations that comes out quit right, but if, for example, an m-cell has only a single line worth of material inside it will be positioned differently to a `l`, `c` or `r` cell or to a `p` or `b` cell with the same content.

For that reason the new implementation does the centering manually: First we check the height of the cell and if that is less or equal to `\ht\strutbox` we assume that this is a single line cell. In that case we don't do any vertical maneuver and simply output the box, i.e., make it behave like a single line p-cell.

We use the height of \strutbox not \@arstrutbox in the comparison, because \box\ar@mcellbox does not have any strut incorporated and if \arraystretch is made very small the test would otherwise incorrectly assume a multi-line cell.

```
\def\ar@align@mcell{%
  \ifdim \ht\ar@mcellbox > \ht\strutbox
```

Otherwise we realign vertically by lowering the box. The question is how much do we need to move down? If there is any \arraystretch in place then the first line will have some unusual height and we don't want to consider that when finding the middle point. So we subtract from the cell height the height of that strut. But of course we want to include the normal height of the first line (which would be something like \ht\strutbox) so we need to add that. On the other hand, when centering around the mid-point of the cell, we also need to account for the depth of the last line (which is nominally something like \dp\strutbox). Both together equals \baselineskip so that is what we add and then lower the cell by half of the resulting value.

```
    \begingroup
     \dimen@\ht\ar@mcellbox
     \advance\dimen@-\ht\@arstrutbox
     \advance\dimen@\baselineskip
     \lower.5\dimen@\box\ar@mcellbox
    \endgroup
  \else % assume one line and align at baseline
    \box\ar@mcellbox
  \fi}
```

(*End of definition for* \ar@align@mcell.)

\@classix *The code below has been replaced long time ago by an extended version further down but the code and its documentation was left here for reference. It is now commented out to avoid confusion.*

In case of class 9 (>–token) we first check if the character class of the last token was 3. In this case we have a user preamble of the form ..>{...}>{...}.. which is not allowed. We only give an error message and continue. So the declarations defined by the first >{...} are ignored.

```
%\def\@classix{\ifnum \@lastchclass = \thr@@
%       \@preamerr \thr@@ \fi
```

Furthermore, we call up \@class10 because afterwards always a new column is started by c, l, r, p, m or b.

```
%       \@classx}
```

(*End of definition for* \@classix.)

\@classviii *The code below has been replaced long time ago by an extended version further down but the code and its documentation was left here for reference. It is now commented out to avoid confusion.*

If the current token is a < the last character class must be 0. In this case it is not necessary to extend the preamble. Otherwise we output an error message, set \@chclass to 6 and call \@classvi. By doing this we achieve that < is treated like !.

```
%\def\@classviii{\ifnum \@lastchclass >\z@
%      \@preamerr 4\@chclass 6 \@classvi \fi}
```

(*End of definition for* \@classviii.)

\@arrayrule There is only one incompatibility with the original definition: the definition of `\@arrayrule`. In the original a line without width[7] is created by multiple insertions of `\hskip .5\arrayrulewidth`. We only insert a vertical line into the preamble. This is done to prevent problems with TEX's main memory when generating tables with many vertical lines in them (especially in the case of floats).

```
\def\@arrayrule{\@addtopreamble \vline}
```

(*End of definition for* `\@arrayrule`.)

\@classvii As a consequence it follows that in case of class 7 (`@` token) the preamble need not to be extended. In the original definition `\@lastchclass` = 1 is treated by inserting `\hskip .5\arrayrulewidth`. We only check if the last token was of class 3 which is forbidden.

```
\def\@classvii{\ifnum \@lastchclass = \thr@@
```

If this is true we output an error message and ignore the declarations stored by the last `>{...}`, because these are overwritten by the argument of `@`.

```
    \@preamerr \thr@@ \fi}
```

(*End of definition for* `\@classvii`.)

\@classvi If the current token is a regular `!` and the last class was 0 or 2 we extend the preamble with `\hskip\col@sep`. If the last token was of class 1 (for instance `|`) we extend with `\hskip \doublerulesep` because the construction `!{...}` has to be treated like `|`.

```
\def\@classvi{\ifcase \@lastchclass
      \@acol \or
      \@addtopreamble{\hskip \doublerulesep}\or
      \@acol \or
```

Now `\@preamerr...` should follow because a user preamble of the form `..>{..}!.` is not allowed. To save memory we call `\@classvii` instead which also does what we want.

```
      \@classvii
```

If `\@lastchclass` is 4 or 5 nothing has to be done. Class 6 to 10 are not possible. So we finish the macro.

```
      \fi}
```

(*End of definition for* `\@classvi`.)

\@classii \@classiii In the case of character classes 2 and 3 (i.e. the argument of `<` or `>`) we only have to store the current token (`\@nextchar`) into the corresponding token register since the preparation and insertion of these registers are done by the macro `\@classz`. This is equivalent to calling `\save@decl` in the case of class 3. To save command identifiers we do this call up in the macro `\@mkpream`.

Class 2 exhibits a more complicated situation: the token registers have already been inserted by `\@classz`. So the value of `\count@` is too high by one. Therefore we decrease `\count@` by 1.

```
\def\@classii{\advance \count@ \m@ne
```

Next we store the current token into the correct token register by calling `\save@decl` and then increase the value of `\count@` again. At this point we can save memory once more (at the cost of time) if we use the macro `\prepnext@tok`.

```
    \save@decl\prepnext@tok}
```

[7] So the space between `cc` and `c|c` is equal.

(*End of definition for* `\@classii` *and* `\@classiii`.)

`\@classv` *The code below has been replaced long time ago by an extended version further down but the code and its documentation was left here for reference. It is now commented out to avoid confusion.*

If the current token is of class 5 then it is an argument of a `@` token. It must be stored into a token register.

```
%\def\@classv{\save@decl
```

We extend the preamble with a command which inserts this token register into the preamble when its construction is finished. The user expects that this argument is worked out in math mode if it was used in an array–environment. Therefore we surround it with `\d@llar...`'s.

```
%    \@addtopreamble{\d@llarbegin\the@toks\the\count@\relax\d@llarend}%
```

Finally we must prepare the next token register.

```
%    \prepnext@tok}
```

(*End of definition for* `\@classv`.)

`\@classi` In the case of class 0 we were able to generate the necessary space between columns by using the macro `\@classx`. Analogously the macro `\@classvi` can be used for class 1.

```
\def\@classi{\@classvi
```

Depending on `\@chnum` a vertical line

```
    \ifcase \@chnum \@arrayrule \or
```

or (in case of `!{...}`) the current token — stored in `\@nextchar` — has to be inserted into the preamble. This corresponds to calling `\@classv`.

```
        \@classv \fi}
```

(*End of definition for* `\@classi`.)

`\@startpbox` `\@startpbox@action` In `\@classz` the macro `\@startpbox` is used. The width of the parbox is passed as an argument. `\vcenter`, `\vtop` or `\vbox` are already in the preamble. So we start with the braces for the wanted box.

```
\def\@startpbox#1{\bgroup
  \color@begingroup
```

The argument is the width of the box. This information has to be assigned to `\hsize`. Then we assign default values to several parameters used in a parbox.

```
    \setlength\hsize{#1}\@arrayparboxrestore
```

Our main problem is to obtain the same distance between succeeding lines of the parbox. We have to remember that the distance between two parboxes should be defined by `\@arstrut`. That means that it can be greater than the distance in a parbox. Therefore it is not enough to set a `\@arstrut` at the beginning and at the end of the parbox. This would dimension the distance between first and second line and the distance between the two last lines of the parbox wrongly. To prevent this we set an invisible rule of height `\@arstrutbox` at the beginning of the parbox. This has no effect on the depth of the first line. At the end of the parbox we set analogously another invisible rule which only affects the depth of the last line. It is necessary to wait inserting this strut until the paragraph actually starts to allow for things like `\parindent` changes via `>{...}`.

```
    \everypar{%
       \vrule \@height \ht\@arstrutbox \@width \z@
       \everypar{}}%
    }
```

Starting with 2.6h, `\@startpbox` is used only during the built-up of the preamble and then changed to `\@startpbox@action` which does the real work. For an explanation of this code see the documentation in `\@mkpream`. We keep the `\@startpbox` definition on top-level in case there are tabular packages that built the preamble differently and expect that it contains `\@startpbox` (longtable at the moment).

```
\let\@startpbox@action\@startpbox
```

(*End of definition for* `\@startpbox` *and* `\@startpbox@action`.)

`\@endpbox` If there are any declarations defined by `>{...}` and `<{...}` they now follow in the macro `\@classz` — the contents of the column in between. So the macro `\@endpbox` must insert the specialstrut mentioned earlier and then close the group opened by `\@startpbox`.

```
\def\@endpbox{\@finalstrut\@arstrutbox \par \color@endgroup \egroup\hfil}
```

(*End of definition for* `\@endpbox`.)

`\@finalstrut` We also have to adjust `\@finalstrut` so that it doesn't back up by a `\baselineskip` when the the current vlist is completely empty. This should go eventually to `ltboxes` but not in a PL.

```
\def\@finalstrut#1{%
  \unskip
  \ifhmode \nobreak
  \else
    \ifnum\lastnodetype>\m@ne
      \vskip-\baselineskip
    \fi
  \fi
  \vrule\@width\z@\@height\z@\@depth\dp#1}
```

(*End of definition for* `\@finalstrut`.)

9 Building and calling `\halign`

`\@array` After we have discussed the macros needed for the evaluation of the user preamble we can define the macro `\@array` which uses these macros to create a `\halign`. It has two arguments. The first one is a position argument which can be `t`, `b` or `c`; the second one describes the wanted preamble, e.g. it has the form `|c|c|c|`.

```
\def\@array[#1]#2{
```

First we define a strut whose size basically corresponds to a normal strut multiplied by the factor `\arraystretch`. This strut is then inserted into every row and enforces a minimal distance between two rows. Nevertheless, when using horizontal lines, large letters (like accented capital letters) still collide with such lines. Therefore at first we add to the height of a normal strut the value of the parameter `\extrarowheight`.

```
  \@tempdima \ht \strutbox
  \advance \@tempdima by\extrarowheight
  \setbox \@arstrutbox \hbox{\vrule
              \@height \arraystretch \@tempdima
              \@depth \arraystretch \dp \strutbox
              \@width \z@}%
```

Before starting a table we have to initialize the variables holding row and column information for cells. We also have locally store the information related to the current cell (if we are already inside a table) so that we can restore it once the inner table is finished. The total number of table columns of the current table is determined in `\tbl_count_table_cols:` but this is called in a group, so local settings do not survive. Thus, to save away the outer value of `\g__tbl_table_cols_tl` we do that also `\tbl_init_cell_data_for_table:`.

```
\tbl_init_cell_data_for_table:
```

Then we open a group, in which the user preamble is evaluated by the macro `\@mkpream`. As we know this must happen locally. This macro creates a preamble for a `\halign` and saves its result globally in the control sequence `\@preamble`.

```
\begingroup
\@mkpream{#2}%
```

Figure out how many columns this table has:

```
\tbl_count_table_cols:
```

We again redefine `\@preamble` so that a call up of `\@preamble` now starts the `\halign`. Thus also the arguments of `>`, `<`, `@` and `!`, saved in the `token` registers are inserted into the preamble. The `\tabskip` at the beginning and end of the preamble is set to `0pt` (in the beginning by the use of `\ialign`). Also the command `\@arstrut` is build in, which inserts the `\@arstrutbox`, defined above. Of course, the opening brace after `\ialign` has to be implicit as it will be closed in `\endarray` or another macro.

The `\noexpand` in front of `\ialign` does no harm in standard LaTeX and was added since some experimental support for using text glyphs in math redefines `\halign` with the result that is becomes expandable with disastrous results in cases like this. In the kernel definition for this macro the problem does not surface because there `\protect` is set (which is not necessary in this implementation as there is no arbitrary user input that can get expanded) and the experimental code made the redefinition robust. Whether this is the right approach is open to question; consider the `\noexpand` a courtesy to allow an unsupported redefinition of a TeX primitive for the moment (as people rely on that experimental code).

```
\xdef\@preamble{
```

`\ialign` in the original definition is replaced by `\ar@ialign` defined below. This does what `\ialign` does but additionally handles the tagging structure for the whole table if necessary.

```
  \noexpand \ar@ialign
  \@halignto
    \bgroup \@arstrut
```

What we have not explained yet is the macro `\@halignto` that was just used. Depending on its replacement text the `\halign` becomes a `\halign to` ⟨*dimen*⟩.

Next, a tagging support socket is inserted adding the start row tag.

```
    \UseTaggingSocket{tbl/row/begin}
```

At the start of the preamble for the first column we call `\tbl_init_cell_data_for_row:` to initialize the cell index data. In later columns this data is updated via `\tbl_update_cell_data:`.

```
    \tbl_init_cell_data_for_row:
    \@preamble
    \tabskip \z@ \cr}
```

Now we close the group again. Thus `\@startpbox` and `\@endpbox` as well as all token registers get their former meaning back.

```
\endgroup
```

To support the `delarray.sty` package we include a hook into this part of the code which is a no-op in the main package.

```
\@arrayleft
```

Now we decide depending on the position argument in which box the `\halign` is to be put. (`\vcenter` may be used because we are in math mode.)

```
\if #1t\vtop \else \if#1b\vbox \else \vcenter \fi \fi
```

Now another implicit opening brace appears; then definitions which shall stay local follow. While constructing the `\@preamble` in `\@mkpream` the `#` sign must be hidden in the macro `\@sharp` which is `\let` to `\relax` at that moment (see definition of `\@mkpream` on page 18). All these now get their actual meaning.

```
\bgroup
\let \@sharp ##\let \protect \relax
```

With the above defined struts we fix down the distance between rows by setting `\lineskip` and `\baselineskip` to 0pt. Since there have to be set `$`'s around every column in the array–environment the parameter `\mathsurround` should also be set to 0pt. This prevents additional space between the rows.

```
\lineskip \z@
\baselineskip \z@
```

Don't use `\m@th` here as that signals to the math tagging code that this is fake math that should not be tagged.

```
\mathsurround \z@
```

Beside, we have to assign a special meaning (which we still have to specify) to the line separator `\\`. We also have to redefine the command `\par` in such a way that empty lines in `\halign` cannot do any damage. We succeed in doing so by choosing something that will disappear when expanding. After that we only have to call up `\@preamble` to start the wanted `\halign`.

```
\let\\\@arraycr \let\tabularnewline\\%
\def\par{\ifnum\currentgrouptype=6~ \else\@@par\fi}%
```

Another socket for tagging. TODO: what about `\arrayleft` above?

```
\UseTaggingSocket{tbl/init}

\@preamble
}
```

(*End of definition for* `\@array`.)

`\ar@ialign` A new command that replaces `\ialign` used previously. `\everycr` is also applied to the `\cr` ending the preamble so we have to program around that.

```
\def\ar@ialign{%

  \everycr{%
    \noalign{%
```

If this `\cr` was at the end of a real row (e.g., not at the end of the table preamble) we have add a row end tag.

```
      \tbl_if_row_was_started:T  { \UseTaggingSocket{tbl/row/end} }
```

The we prepare for the next row.

```
      \tbl_update_cell_data_for_next_row:
    }%
  }%
  \tabskip\z@skip\halign}
```

(*End of definition for* `\ar@ialign`.)

`\arraybackslash` Restore `\\` for use in array and tabular environment (after `\raggedright` etc.).

```
\def\arraybackslash{\let\\\tabularnewline}
```

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

`\extrarowheight` The dimen parameter used above also needs to be allocated. As a default value we use 0pt, to ensure compatibility with standard LaTeX.

```
\newdimen \extrarowheight
\extrarowheight=0pt
```

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

`\@arstrut` Now the insertion of `\@arstrutbox` through `\@arstut` is easy since we know exactly in which mode TeX is while working on the `\halign` preamble.

```
\def\@arstrut{\unhcopy\@arstrutbox}
```

(*End of definition for* `\@arstrut`.)

10 The line separator \\

`\@arraycr` In the macro `\@array` the line separator `\\` is `\let` to the command `\@arraycr`.

```
\protected\def\@arraycr {
```

Add code that figures out if the current table row is incomplete (not enough `&`s). It can then do extra actions, such as inserting missing cell tags.

```
  \tbl_count_missing_cells:n {@arraycr}
```

TODO: maybe this is also the right place to add a socket that could be used to actually enter missing cells instead of just adding tagging structures for them later. This would be optional but in many cases it would be the right thing to do (for example if tables contain vertical lines or similar visual structures that require fully specified rows.

We then start a special brace which I have directly copied from the original definition. It is necessary, because the `\futurelet` in `\@ifnextchar` might expand a following `&` token in a construction like `\\ &`. This would otherwise end the alignment template at a wrong time. On the other hand we have to be careful to avoid producing a real group, i.e. `{}`, because the command will also be used for the array environment, i.e. in math mode. In that case an extra `{}` would produce an ord atom which could mess up the spacing. For this reason we use a combination that does not really produce a group at all but modifies the master counter so that a `&` will not be considered belonging to the current `\halign` while we are looking for a `*` or `[`. For further information see [2, Appendix D].

```
  \iffalse{\fi\ifnum 0=‘}\fi
```

Then we test whether the user is using the star form and ignore a possible star (I also disagree with this procedure, because a star does not make any sense here).

```
  \@ifstar \@xarraycr \@xarraycr}
```

(*End of definition for* `\@arraycr`.)

`\@xarraycr` In the command `\@xarraycr` we test if an optional argument exists.

```
\def\@xarraycr{\@ifnextchar [%
```

If it does, we branch out into the macro `\@argarraycr` if not we close the special brace (mentioned above) and end the row of the `\halign` with a `\cr`.

```
  \@argarraycr {\ifnum 0=`{}\fi\cr}}
```

(*End of definition for* `\@xarraycr`.)

`\@argarraycr` If additional space is requested by the user this case is treated in the macro `\@argarraycr`. First we close the special brace and then we test if the additional space is positive.

```
\def\@argarraycr[#1]{\ifnum0=`{}\fi\ifdim #1>\z@
```

If this is the case we create an invisible vertical rule with depth `\dp\@arstutbox`+⟨*wanted space*⟩. Thus we achieve that all vertical lines specified in the user preamble by a `|` are now generally drawn. Then the row ends with a `\cr`.

If the space is negative we end the row at once with a `\cr` and move back up with a `\vskip`.

While testing these macros I found out that the `\endtemplate` created by `\cr` and `&` is something like an `\outer` primitive and therefore it should not appear in incomplete `\if` statements. Thus the following solution was chosen which hides the `\cr` in other macros when TEX is skipping conditional text.

```
  \expandafter\@xargarraycr\else
  \expandafter\@yargarraycr\fi{#1}}
```

(*End of definition for* `\@argarraycr`.)

`\@xargarraycr` The following macros were already explained above.
`\@yargarraycr`

```
\def\@xargarraycr#1{\unskip\gdef\do@row@strut
 {\@tempdima #1\advance\@tempdima \dp\@arstrutbox
   \vrule \@depth\@tempdima \@width\z@\global\let\do@row@strut\relax}%
```

If the last column is a `\multicolumn` cell then we need to insert the row strut now as it isn't inside the template (as that got `\omit`ted).

```
   \ifnum\@multicnt >\z@ \do@row@strut \fi
   \cr}
\let\do@row@strut\relax
```

`\@yargarraycr` is the same as in the LATEX kernel (depending on the date of the kernel with one of the two definitions below). We therefore do not define it again.

```
%\def\@yargarraycr#1{\cr\noalign{\@vspace@calcify{#1}}} % 2020-10-01
%\def\@yargarraycr#1{\cr\noalign{\vskip #1}}
```

(*End of definition for* `\@xargarraycr` *and* `\@yargarraycr`.)

11 Spanning several columns

\multicolumn

If several columns should be held together with a special format the command \multicolumn must be used. It has three arguments: the number of columns to be covered; the format for the result column and the actual column entry.

```
\long\def\multicolumn#1#2#3{%
```

First we combine the given number of columns into a single one; then we start a new block so that the following definition is kept local.

```
  \multispan{#1}\begingroup
```

For tagging support we have to solve two problems: \multicolumn must handle the row begin if it is used there, and it must save the numbers of cells it spans so that we can add a suitable ColSpan attribute. We do this in the next macro (which in turn calls the tbl/row/begin socket, if necessary).

```
  \tbl_update_multicolumn_cell_data:n {#1}
```

Since a \multicolumn should only describe the format of a result column, we redefine \@addamp in such a way that one gets an error message if one uses more than one c, l, r, p, m or b in the second argument. One should consider that this definition is local to the build-up of the preamble; an array– or tabular–environment in the third argument of the \multicolumn is therefore worked through correctly as well.

```
  \def\@addamp{\if@firstamp \@firstampfalse \else
                \@preamerr 5\fi}%
```

Then we evaluate the second argument with the help of \@mkpream. Now we still have to insert the contents of the token register into the \@preamble, i.e. we have to say \xdef\@preamble{\@preamble}. This is achieved shorter by writing:

```
  \@mkpream{#2}\@addtopreamble\@empty
```

After the \@preamble is created we forget all local definitions and occupations of the token registers.

```
  \endgroup
```

Now we update the colspan attribute. This needs setting after the group as it is hidden inside the plug in \insert@column.

```
  \UseTaggingSocket{tbl/colspan}{#1}%
```

In the special situation of \multicolumn \@preamble is not needed as preamble for a \halign but it is directly inserted into our table. Thus instead of \sharp there has to be the column entry (#3) wanted by the user.

```
  \def\@sharp{#3}%
```

Now we can pass the \@preamble to TEX . For safety we start with an \@arstrut. This should usually be in the template for the first column however we do not know if this template was overwritten by our \multicolumn. We also add a \null at the right end to prevent any following \unskip (for example from \\[..]) to remove the \tabcolsep.

```
  \@arstrut \@preamble
  \null
  \ignorespaces}
```

(*End of definition for* \multicolumn.)

12 The Environment Definitions

After these preparations we are able to define the environments. They only differ in the initializations of `\d@llar...`, `\col@sep` and `\@halignto`.

`\@halignto` `\d@llarbegin` `\d@llarend`

`\d@llar` has to be locally assigned since otherwise nested tabular and array environments (via `\multicolumn`) are impossible. For 25 years or so `\@halignto` was set globally (to save space on the save stack, but that was a mistake: if there is a tabular in the output routine (e.g., in the running header) then that tabular is able overwrite the `\@halignto` setting of a tabular in the main text resulting in a very weird error. When the new font selection scheme is in force we have to we surround all `\halign` entries with braces. See remarks in TUGboat 10#2. Actually we are going to use `\begingroup` and `\endgroup`. However, this is only necessary when we are in text mode. In math the surrounding dollar signs will already serve as the necessary extra grouping level. Therefore we switch the settings of `\d@llarbegin` and `\d@llarend` between groups and dollar signs.

```
\let\d@llarbegin\begingroup
\let\d@llarend\endgroup
```

(*End of definition for* `\@halignto`, `\d@llarbegin`, *and* `\d@llarend`.)

`\array`

Our new definition of `\array` then reads: In math mode we additionally add also the tagging sockets for luamml. . The `math/luamml/array/finalizecol` takes as argument a number (0, 1, 2) that should represent the alignment (c,l,r). TODO: instead of `\the\@chnum` a proper tl should be used. TODO: the w and W columntypes must be redefined - currently they insert stray mtext columns.

```
\def\array{\col@sep\arraycolsep
  \def\d@llarbegin{$\tag_socket_use:n { math/luamml/array/initcol }}
  \def\d@llarend
   {
     \tag_socket_use:nn { math/luamml/save/nn }{{}{mtd}}
     $
     \tag_socket_use:nn { math/luamml/array/finalizecol}{\the\@chnum}
   }
  \def\@halignto{}%
```

Since there might be an optional argument we call another macro which is also used by the other environments.

```
  \@tabarray}
```

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

`\@tabarray`

The code below has been replaced long time ago by an extended version further down but the code and its documentation was left here for reference. It is now commented out to avoid confusion.

This macro tests for a optional bracket and then calls up `\@array` or `\@array[c]` (as default).

```
%\def\@tabarray{\@ifnextchar[{\@array}{\@array[c]}}
```

(*End of definition for* `\@tabarray`.)

`\tabular` `\tabular*`

The environments tabular and tabular∗ differ only in the initialization of the command `\@halignto`. Therefore we define

```
\def\tabular{\def\@halignto{}\@tabular}
```

and analogously for the star form. We evaluate the argument first using `\setlength` so that users of the `calc` package can write code like `\begin{tabular*}{(\columnwidth-1cm)/2}...`

```
\expandafter\def\csname tabular*\endcsname#1{%
      \setlength\dimen@{#1}%
      \edef\@halignto{to\the\dimen@}\@tabular}
```

(*End of definition for* `\tabular` *and* `\tabular*`.)

`\@tabular` The rest of the job is carried out by the `\@tabular` macro:

```
\providecommand\UseMathForPositioningText{}
\providecommand\@kernel@tabular@init{}
\def\@tabular{%
```

First of all we have to make sure that we start out in hmode. Otherwise we might find our table dangling by itself on a line.

```
  \leavevmode
```

Now that we know we are in hmode we can add the start tag for the whole table.

```
  \UseTaggingSocket{tbl/hmode/begin}%
```

It should be taken into consideration that the macro `\@array` must be called in math mode. Therefore we open a box, insert a $ and then assign the correct values to `\col@sep` and `\d@llar...`.

```
  \hbox \bgroup
    \UseMathForPositioningText $%
    \col@sep\tabcolsep
    \let\d@llarbegin\begingroup
    \let\d@llarend\endgroup

   \@kernel@tabular@init
```

Now everything tabular specific is done and we are able to call the `\@tabarray` macro.

```
  \@tabarray}
```

(*End of definition for* `\@tabular`.)

`\endarray` When the processing of array is finished we have to close the `\halign` and afterwards the surrounding box selected by `\@array`. To save token space we then redefine `\@preamble` because its replacement text isn't longer needed.

To handle cell indexes, we do not use `\crcr` but a variant that also handles missing cells as necessary.

```
\def\endarray {
  \tbl_crcr:n{endarray}
  \tag_socket_use_expandable:n { math/luamml/array/save }
  \egroup
  \tag_socket_use:n { tbl/finalize }
```

If tables are nested into another then it is necessary to restore information about the cell the inner table started in. Otherwise, the cell index data structures reflect the status in the outer table as they are globally manipulated. We restore in all cases even if we are not in a nesting situation as that makes the code simpler (and probably faster).

`\endtabular` and `\endtabular*` inherit from `\endarray` so we only need to change that. `tabularx` uses a similar method.

```
  \tbl_restore_outer_cell_data:
```

```
  \egroup
  \tag_socket_use:n { math/luamml/array/finalize }
  \@arrayright \gdef\@preamble{}%
}
```

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

`\endtabular` `\endtabular*` To end a tabular or tabular∗ environment we call up `\endarray`, close the math mode and then the surrounding `\hbox`. This math mode around the tabular should not be surrounded by any `\mathsurround` so we cancel that with `\m@th`.

```
\def\endtabular{\endarray\m@th $\egroup
  \UseTaggingSocket{tbl/hmode/end}%
}
\expandafter\let\csname endtabular*\endcsname=\endtabular
```

(*End of definition for* `\endtabular` *and* `\endtabular*`.)

13 Last minute definitions

If this file is used as a package file we should `\let` all macros to `\relax` that were used in the original but are no longer necessary.

```
\let\@ampacol=\relax        \let\@expast=\relax
\let\@arrayclassiv=\relax   \let\@arrayclassz=\relax
\let\@tabclassiv=\relax     \let\@tabclassz=\relax
\let\@arrayacol=\relax      \let\@tabacol=\relax
\let\@tabularcr=\relax      \let\@@endpbox=\relax
\let\@argtabularcr=\relax   \let\@xtabularcr=\relax
```

`\@preamerr` We also have to redefine the error routine `\@preamerr` since new kind of errors are possible. The code for this macro is not perfect yet; it still needs too much memory.

```
\ExplSyntaxOff
\def\@preamerr#1{\def\@tempd{{..} at wrong position: }%
   \PackageError{array}{%
   \ifcase #1 Illegal pream-token (\@nextchar): `c' used\or %0
    Missing arg: token ignored\or                          %1
    Empty preamble: `l' used\or                            %2
    >\@tempd token ignored\or                              %3
    <\@tempd changed to !{..}\or                           %4
    Only one column-spec. allowed.\fi}\@ehc}               %5
```

(*End of definition for* `\@preamerr`.)

14 Defining your own column specifiers[8]

`\newcolumn` In `newarray.sty` the macro for specifying new columns was named `\newcolumn`. When the functionality was added to `array.sty` the command was renamed `\newcolumntype`.

[8] The code and the documentation in this section was written by David. So far only the code from newarray was plugged into array so that some parts of the documentation still claim that this is newarray and even worse, some parts of the code are unnecessarily doubled. This will go away in a future release. For the moment we thought it would be more important to bring both packages together.

Initially both names were supported, but now (In versions of this package distributed for LaTeX 2_ε) the old name is not defined.

```
⟨*ncols⟩
```

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

`\newcolumntype` As described above, the `\newcolumntype` macro gives users the chance to define letters, to be used in the same way as the primitive column specifiers, 'c' 'p' etc.

```
\def\newcolumntype#1{%
```

`\NC@char` was added in V2.01 so that active characters, like `@` in AMSLaTeX may be used. This trick was stolen from `array.sty` 2.0h. Note that we need to use the possibly active token, `#1`, in several places, as that is the token that actually appears in the preamble argument.

```
  \edef\NC@char{\string#1}%
```

First we check whether there is already a definition for this column. Unlike `\newcommand` we give a warning rather than an error if it is defined. If it is a new column, add `\NC@do` ⟨*column*⟩ to the list `\NC@list`.

```
  \@ifundefined{NC@find@\NC@char}%
    {\@tfor\next:=<>clrmbp@!|\do
      {%
```

We use `\noexpand` on the tokens from the list in case one or the other (typically `@`, `!` or `|`) has been made active.

```
        \if\expandafter\noexpand\next\NC@char
        \PackageWarning{array}%
                        {Redefining primitive column \NC@char}\fi}%
     \NC@list\expandafter{\the\NC@list\NC@do#1}}%
    {\PackageWarning{array}{Column \NC@char\space is already defined}}%
```

Now we define a macro with an argument delimited by the new column specifier, this is used to find occurrences of this specifier in the user preamble.

```
  \@namedef{NC@find@\NC@char}##1#1{\NC@{##1}}%
```

If an optional argument was not given, give a default argument of 0.

```
  \@ifnextchar[{\newcol@{\NC@char}}{\newcol@{\NC@char}[0]}}
\ExplSyntaxOn
```

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

`\newcol@` We can now define the macro which does the rewriting, `\@reargdef` takes the same arguments as `\newcommand`, but does not check that the command is new. For a column, say 'D' with one argument, define a command `\NC@rewrite@D` with one argument, which recursively calls `\NC@find` on the user preamble after replacing the first token or group with the replacement text specified in the `\newcolumntype` command. `\NC@find` will find the next occurrence of 'D' as it will be `\let` equal to `\NC@find@D` by `\NC@do`.

```
\def\newcol@#1[#2]#3{\expandafter\@reargdef
      \csname NC@rewrite@#1\endcsname[#2]{\NC@find#3}}
```

(*End of definition for* `\newcol@`.)

\NC@ Having found an occurrence of the new column, save the preamble before the column in \@temptokena, then check to see if we are at the end of the preamble. (A dummy occurrence of the column specifier will be placed at the end of the preamble by \NC@do.

```
\def\NC@#1{%
  \@temptokena\expandafter{\the\@temptokena#1}\futurelet\next\NC@ifend}
```

(*End of definition for* \NC@.)

\NC@ifend We can tell that we are at the end as \NC@do will place a \relax after the dummy column.

```
\def\NC@ifend{%
```

If we are at the end, do nothing. (The whole preamble will now be in \@temptokena.)

```
  \ifx\next\relax
```

Otherwise set the flag \if@tempswa, and rewrite the column. \expandafter introduced 1n V2.01

```
    \else\@tempswatrue\expandafter\NC@rewrite\fi}
```

(*End of definition for* \NC@ifend.)

\NC@do If the user has specified 'C' and 'L' as new columns, the list of rewrites (in the token register \NC@list) will look like \NC@do * \NC@do C \NC@do L. So we need to define \NC@do as a one argument macro which initializes the rewriting of the specified column. Let us assume that 'C' is the argument.

```
\def\NC@do#1{%
```

First we let \NC@rewrite and \NC@find be \NC@rewrite@C and \NC@find@C respectively.

```
  \expandafter\let\expandafter\NC@rewrite
    \csname NC@rewrite@\string#1\endcsname
  \expandafter\let\expandafter\NC@find
    \csname NC@find@\string#1\endcsname
```

Clear the token register \@temptokena after putting the present contents of the register in front of the token \NC@find. At the end we place the tokens 'C\relax' which \NC@ifend will use to detect the end of the user preamble.

```
  \expandafter\@temptokena\expandafter{\expandafter}%
        \expandafter\NC@find\the\@temptokena#1\relax}
```

(*End of definition for* \NC@do.)

\showcols This macro is useful for debugging \newcolumntype specifications, it is the equivalent of the primitive \show command for macro definitions. All we need to do is locally redefine \NC@do to take its argument (say 'C') and then \show the (slightly modified) definition of \NC@rewrite@C. Actually as the list always starts off with \NC@do * and we do not want to print the definition of the ∗-form, define \NC@do to throw away the first item in the list, and then redefine itself to print the rest of the definitions.

```
\def\showcols{{\def\NC@do##1{\let\NC@do\NC@show}\the\NC@list}}
```

(*End of definition for* \showcols.)

\NC@show If the column 'C' is defined as above, then `\show\NC@rewrite@C` would output `\long macro: ->\NC@find >{$}c<{$}`. We want to strip the `long macro: ->` and the `\NC@find`. So first we use `\meaning` and then apply the macro `\NC@strip` to the tokens so produced and then `\typeout` the required string.

```
\def\NC@show#1{%
  \typeout{Column~ #1\expandafter\expandafter\expandafter\NC@strip
  \expandafter\meaning\csname NC@rewrite@#1\endcsname\@@}}
```

(*End of definition for* `\NC@show`.)

\NC@strip Delimit the arguments to `\NC@strip` with ':', '->', a space, and `\@@@@` to pull out the required parts of the output from `\meaning`.

```
\ExplSyntaxOff
\def\NC@strip#1:#2->#3 #4\@@{#2 -> #4}
\ExplSyntaxOn
```

(*End of definition for* `\NC@strip`.)

\NC@list Allocate the token register used for the rewrite list.

```
\newtoks\NC@list
```

(*End of definition for* `\NC@list`.)

14.1 The ∗–form

We view the ∗-form as a slight generalization of the system described in the previous subsection. The idea is to define a ∗ column by a command of the form:

```
\newcolumntype{*}[2]{%
   \count@=#1\ifnum\count@>0
      \advance\count@ by -1 #2*{\count@}{#2}\fi}
```

\NC@rewrite@* This does not work however as `\newcolumntype` takes great care not to expand anything in the preamble, and so the `\if` is never expanded. `\newcolumntype` sets up various other parts of the rewrite correctly though so we can define:

```
\newcolumntype{*}[2]{}
```

Now we must correct the definition of `\NC@rewrite@*`. The following is probably more efficient than a direct translation of the idea sketched above, we do not need to put a ∗ in the preamble and call the rewrite recursively, we can just put `#1` copies of `#2` into `\@temptokena`. (Nested ∗ forms will be expanded when the whole rewrite list is expanded again, see `\@mkpream`)

```
\long\@namedef{NC@rewrite@*}#1#2{%
```

Store the number.

```
  \count@#1\relax
```

Put `#1` copies of `#2` in the token register.

```
  \loop
  \ifnum\count@>\z@
  \advance\count@\m@ne
  \@temptokena\expandafter{\the\@temptokena#2}%
  \repeat
```

`\NC@do` will ensure that `\NC@find` is `\let` equal to `\NC@find@*`.

```
  \NC@find}
```

(*End of definition for* `\NC@rewrite@*`.)

14.2 Modifications to internal macros of `array.sty`

\@xexpast
\@xexnoop

These macros are used to expand ∗-forms in `array.sty`. `\let` them to `\relax` to save space.

```
\let\@xexpast\relax
\let\@xexnoop\relax
```

(*End of definition for* `\@xexpast` *and* `\@xexnoop`.)

\save@decl

We do not assume that the token register is free, we add the new declarations to the front of the register. This is to allow user preambles of the form, `>{foo}>{bar}...` Users are not encouraged to enter such expressions directly, but they may result from the rewriting of `\newcolumntype`'s.

```
\def\save@decl{\toks \count@ = \expandafter\expandafter\expandafter
                   {\expandafter\@nextchar\the\toks\count@}}
```

(*End of definition for* `\save@decl`.)

\@mkpream

The main modification to `\@mkpream` is to replace the call to `\@xexpast` (which expanded ∗-forms) by a loop which expands all `\newcolumntype` specifiers.

```
\ExplSyntaxOff % really oldstyle using \@tfor :=
\def\@mkpream#1{\gdef\@preamble{}\@lastchclass 4 \@firstamptrue
   \let\@sharp\relax
```

The `\@startpbox` (which is called for `p`, `m` or `b` columns) receives a user supplied argument: the width of the paragraph-column. Normally that is something harmless like a length or a simple length expression, but with the calc package involved it could break under an `\edef` operation, which is how the preamble is constructed. We now make use of `\unexpanded` here to prevent that. The `\expandafter` gymnastics are necessary to expand the `#1` exactly once (since it will get `\@nextchar` as its value and need its content). However, once added to the preamble, no further expansion should happen if the preamble is extended. Therefore we change from `\@startpbox` to `\@startpbox@action`. The latter then has a trivial definition and expands (including its argument) to itself while the preamble is being built. Outside of this preamble build up `\@startpbox@action` does the actual work.

```
   \def\@startpbox##1{\unexpanded\expandafter{\expandafter
                        \@startpbox@action\expandafter{##1}}}%
   \def\@startpbox@action##1{\unexpanded{\@startpbox@action{##1}}}%
   \let\@endpbox\relax
   \let\do@row@strut\relax
   \let\ar@align@mcell\relax
```

Now we remove possible ∗-forms and user-defined column specifiers in the user preamble by repeatedly executing the list `\NC@list` until the re-writes have no more effect. The expanded preamble will then be in the token register `\@temptokena`. Actually we need to know at this point that this is not `\toks0`.

```
   \@temptokena{#1}\@tempswatrue
   \@whilesw\if@tempswa\fi{\@tempswafalse\the\NC@list}%
```

Afterwards we initialize all registers and macros, that we need for the build-up of the preamble.

```
   \count@\m@ne
   \let\the@toks\relax
   \prepnext@tok
```

Having expanded all tokens defined using `\newcolumntype` (including `*`), we evaluate the remaining tokens, which are saved in `\@temptokena`. We use the LaTeX–macro `\@tfor` to inspect each token in turn.

```
\expandafter \@tfor \expandafter \@nextchar
 \expandafter :\expandafter =\the\@temptokena \do
```

`\@testpatch` does not take an argument since `array.sty` 2.0h.

```
{\@testpach
\ifcase \@chclass \@classz \or \@classi \or \@classii
  \or \save@decl \or \or \@classv \or \@classvi
  \or \@classvii \or \@classviii
```

In `newarray.sty` class 9 is equivalent to class 10.

```
  \or \@classx
  \or \@classx \fi
\@lastchclass\@chclass}%
\ifcase\@lastchclass
\@acol \or
\or
\@acol \or
\@preamerr \thr@@ \or
\@preamerr \tw@ \@addtopreamble\@sharp \or
\or
\else  \@preamerr \@ne \fi
\def\the@toks{\the\toks}}
\ExplSyntaxOn
```

(*End of definition for* `\@mkpream`.)

`\@classix` `array.sty` does not allow repeated `>` declarations for the same column. This is allowed in `newarray.sty` as documented in the introduction. Removing the test for this case makes class 9 equivalent to class 10, and so this macro is redundant. It is `\let` to `\relax` to save space.

```
\let\@classix\relax
```

(*End of definition for* `\@classix`.)

`\@classviii` In `newarray.sty` explicitly allow class 2, as repeated `<` expressions are accepted by this package.

```
\def\@classviii{\ifnum \@lastchclass >\z@\ifnum\@lastchclass=\tw@\else
      \@preamerr 4\@chclass 6 \@classvi \fi\fi}
```

(*End of definition for* `\@classviii`.)

`\@classv` Class 5 is `@`-expressions (and is also called by class 1) This macro was incorrect in Version 1. Now we do not expand the `@`-expression, but instead explicitly replace an `\extracolsep` command by an assignment to `\tabskip` by a method similar to the `\newcolumntype` system described above. `\d@llarbegin` `\d@llarend` were introduced in V2.01 to match `array.sty` 2.0h.

```
\def\@classv{\save@decl
   \expandafter\NC@ecs\@nextchar\extracolsep{}\extracolsep\@@__tbl
   \@addtopreamble{\d@llarbegin\the@toks\the\count@\relax\d@llarend}%
   \prepnext@tok}
```

(*End of definition for* `\@classv`.)

\NC@ecs Rewrite the first occurrence of `\extracolsep{1in}` to `\tabskip1in\relax`. As a side effect discard any tokens after a second `\extracolsep`, there is no point in the user entering two of these commands anyway, so this is not really a restriction.

```
\def\NC@ecs#1\extracolsep#2#3\extracolsep#4\@@__tbl{\def\@tempa{#2}%
  \ifx\@tempa\@empty\else\toks\count@={#1\tabskip#2\relax#3}\fi}
⟨/ncols⟩
```

(*End of definition for* `\NC@ecs`.)

14.3 Support for the `delarray.sty`

The `delarray.sty` package extends the array syntax by supporting the notation of delimiters. To this end we extend the array parsing mechanism to include a hook which can be used by this (or another) package to do some additional parsing.

\@tabarray This macro tests for an optional bracket and then calls up `\@@@@array` or `\@@@@array[c]` (as default).

```
⟨*package⟩
\def\@tabarray{\@ifnextchar[{\@@array}{\@@array[c]}}
```

(*End of definition for* `\@tabarray`.)

\@@array This macro tests could then test an optional delimiter before the left brace of the main preamble argument. Here in the main package it simply is let to be `\@array`.

```
\let\@@array\@array
```

(*End of definition for* `\@@array`.)

\@arrrayleft \@arrayright We have to declare the hook we put into `\@array` above. A similar hook `\@arrayright` will be inserted into the `\endarray` to gain control. Both defaults to empty.

```
\let\@arrayleft\@empty
\let\@arrayright\@empty
```

(*End of definition for* `\@arrrayleft` *and* `\@arrayright`.)

14.4 Support for `\firsthline` and `\lasthline`

The Companion [1, p.137] suggests two additional commands to control the alignments in case of tabulars with horizontal lines. They are now added to this package.

\extratabsurround The extra space around a table when `\firsthline` or `\lasthline` are used.

```
\newlength{\extratabsurround}
\setlength{\extratabsurround}{2pt}
```

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

\backup@length This register will be used internally by `\firsthline` and `\lasthline`.

```
\newlength{\backup@length}
```

(*End of definition for* `\backup@length`.)

\firsthline This code can probably be improved but for the moment it should serve.

We start by producing a single tabular row without any visible content that will produce the external reference point in case [t] is used. We need to suppress the \tabcolsep in the \multicolumn in case there wasn't any in the real column.

```
\newcommand{\firsthline}{%
  \multicolumn1{@{}c@{}}{%
```

Within this row we calculate \backup@length to be the height plus depth of a standard line. In addition we have to add the width of the \hline, something that was forgotten in the original definition.

```
    \global\backup@length\ht\@arstrutbox
    \global\advance\backup@length\dp\@arstrutbox
    \global\advance\backup@length\arrayrulewidth
```

Finally we do want to make the height of this first line be a bit larger than usual, for this we place the standard array strut into it but raised by \extratabsurround

```
     \raise\extratabsurround\copy\@arstrutbox
```

And we should also cancel the guard otherwise we end up with two.

```
     \kern-1sp%
```

Having done all this we end the line and back up by the value of \backup@length and then finally place our \hline. This should place the line exactly at the right place but keep the reference point of the whole tabular at the baseline of the first row.

```
    }\\[-\backup@length]\hline
}
```

(*End of definition for* \firsthline.)

\lasthline For \lasthline the situation is even worse and I got it completely wrong initially.

The problem in this case is that if the optional argument [b] is used we do want the reference point of the tabular be at the baseline of the last row but at the same time do want the depth of this last line increased by \extratabsurround without changing the placement \hline.

We start by placing the rule followed by an invisible row. We need to suppress the \tabcolsep in the multicol in case there wasn't any in the real column.

```
\newcommand{\lasthline}{\hline\multicolumn1{@{}c@{}}{%
```

We now calculate \backup@length to be the height and depth of two lines plus the width of the rule.

```
    \global\backup@length2\ht\@arstrutbox
    \global\advance\backup@length2\dp\@arstrutbox
    \global\advance\backup@length\arrayrulewidth
```

This will bring us back to the baseline of the second last row:

```
    }\\[-\backup@length]%
```

Thus if we now add another invisible row the reference point of that row will be at the baseline of the last row (and will be the reference for the whole tabular). Since this row is invisible we can enlarge its depth by the desired amount.

```
    \multicolumn1{@{}c@{}}{%
       \lower\extratabsurround\copy\@arstrutbox
       \kern-1sp%
       }%
}
```

(*End of definition for* \lasthline.)

14.5 Getting the spacing around rules right

Beside a larger functionality `array.sty` has one important difference to the standard `tabular` and `array` environments: horizontal and vertical rules make a table larger or wider, e.g., `\doublerulesep` really denotes the space between two rules and isn't measured from the middle of the rules.

`\@xhline` For vertical rules this is implemented by the definitions above, for horizontal rules we have to take out the backspace.

```
\CheckCommand*\@xhline{\ifx\reserved@a\hline
                \vskip\doublerulesep
                \vskip-\arrayrulewidth
              \fi
      \ifnum0=‘{\fi}}
\renewcommand*\@xhline{\ifx\reserved@a\hline
                \vskip\doublerulesep
              \fi
      \ifnum0=‘{\fi}}
⟨/package⟩
```

(*End of definition for* `\@xhline`.)

14.6 Implementing column types `w` and `W`

In TugBoat 38/2 an extension was presented that implemented two additional column types `w` and `W`. These have now been added to the package itself.

`\ar@cellbox` For `w` and `W` column types we need a box to temporarily hold the cell content.

```
\newsavebox\ar@cellbox
```

(*End of definition for* `\ar@cellbox`.)

`\newcolumntype␣w` The `w` column type has two arguments: the first holds the alignment which is either `l`, `c`, or `r` and the second is the nominal width of the column.

```
\newcolumntype{w}[2]{%
```

Before the cell content we start an `lrbox`-environment to collect the cell material into the previously allocated box `\ar@cellbox`. We add `\d@llarbegin` (and later `\d@llarend`) so that the content is typeset in math mode if we are in an `array` environment.

```
  >{\begin{lrbox}\ar@cellbox\d@llarbegin}%
```

Then comes a specifier for the cell content. We use `c`, but that doesn't matter as in the end we will always put a box of a specific width (`#2`) into the cells of that column, so `l` or `r` would give the same result. There is only a difference if there are also very wide `\multicolumn` rows overwriting the setting in which case `c` seems to be slightly better.

```
  c%
```

At the end of the cell we end the `lrbox` environment so that all of the cell content is now in box `\ar@cellbox`. As a final step we put that box into a `\makebox` using the optional arguments of that command to achieve the correct width and the desired alignment within that width. We unbox the collected material so that any stretchable glue inside can interact with the alignment.

```
  <{\d@llarend \end{lrbox}%
    \makebox[#2][#1]{\unhbox\ar@cellbox}}}
```

(*End of definition for* `\newcolumntype w`.)

`\newcolumntype␣W` The `W` is similar but in this case we want a warning if the cell content is too wide.

```
\newcolumntype{W}[2]
  {>{\begin{lrbox}\ar@cellbox\d@llarbegin}%
   c%
   <{\d@llarend\end{lrbox}%
   \let\hss\hfil
   \makebox[#2][#1]{\unhbox\ar@cellbox}}}
```

This is a bit sneaky, as it temporarily disables `\hss`, but given that we know what goes into that box it should be sufficient.

(*End of definition for* `\newcolumntype W`.)

14.7 Handling `\cline`

In the past array did not have to concern itself with `\cline` but simply used the definition already provided in the kernel. However, for tagged PDF output this definition is insufficient, because it causes incorrect row counting and the rules it generates would need to be marked as artifacts. We therefore update it here.

`\@cline` Tagging support for `\cline`

```
\ExplSyntaxOn
\def\@cline#1-#2\@nil{
  \omit
  \@multicnt#1
  \advance\@multispan\m@ne
  \ifnum\@multicnt=\@ne\@firstofone{&\omit}\fi
  \@multicnt#2
  \advance\@multicnt-#1
  \advance\@multispan\@ne
```

The rule needs artifact tagging in tagged PDF.

```
  \UseTaggingSocket{tbl/leaders/begin}
  \leaders\hrule\@height\arrayrulewidth\hfill
  \UseTaggingSocket{tbl/leaders/end}
```

To the row counting the above appears like an extra row, so we have to correct the count.

```
  \tbl_gdecr_row_count:
  \cr
  \noalign{\vskip-\arrayrulewidth}
}
\ExplSyntaxOff
```

(*End of definition for* `\@cline`.)

```
\ExplSyntaxOff
```

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

Change History

v1.0b

General: ‘@classi (faster), ‘@classvi (new) A in preamble means && in ‘halign. 1

v1.1a

General: New concept: preamblechar: c,l,r,C,L,R,A,p,t,|,@,! 1

v1.1b

General: Again p like original LaTeX and z for centered ‘parbox. 1

v1.2a

General: Completely new implementation. 1

v1.2b

General: | does no longer generate space at start or end of the preamble. Otherwise ‘hline is too long. 1

Enlarged ‘@arstrutbox by 1pt (Test-Impl) with dimen ‘@strutheight. 1

v1.2c

General: Enlarged ‘@arstrutbox by ‘extrarowheight. Thus you may avoid large characters to overprint a ‘hline. 1

Introduced ‘m@th in ‘@array to allow non-zero values of ‘mathsurround. 1

New dimen parameter ‘extrarowheight (default: 0pt). . . . 1

v1.2d

General: Completed the documentation. 1

v1.2e

General: Bug fixed: A at start of preamble resulted in an error since ‘@mkpream generated ‘@arstrut & ... as a preamble. 1

v1.2f

General: ‘@testpach documented. . . . 1

v1.3a

General: Again a new implementation, with a new concept (cf. the documentation). 1

v1.3b

General: ‘@decl expands now into ‘@empty, i.e., it disappears when the preamble is generated, except when the user specifies A{} or B{}. 1

v1.4a

General: Test implementation of use of token registers in order to do without ‘protect. 1

v1.4b

General: Changed erroneous class numbers: 5 -> 6 6 -> 7 7 -> 5 Corresponding changes in the macros. 1

v1.4c

General: Everything except p,z now works with token registers. 1

References

[1] M. Goossens, F. Mittelbach and A. Samarin. The LaTeX Companion. Addison-Wesley, Reading, Massachusetts, 1994.

[2] D. E. Knuth. The TeXbook (Computers & Typesetting Volume A). Addison-Wesley, Reading, Massachusetts, 1986.

[3] L. Lamport. LaTeX — A Document Preparation System. Addison-Wesley, Reading, Massachusetts, 1986.