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

Frank Mittelbach David Carlisle†

Printed July 3, 1995

Abstract

This article describes a new 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.

At the same time it shows a new — and in our opinion sensible — way of documenting TeX-macros: This article is the package–file that is to be used. All we need in addition to it is a short TeX–program which visualizes the comments and puts the definitions in verbatim mode.

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 Lamport [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.
- 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`.

*This file has version number v2.3c, last revised 1995/04/23.

†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 `@{...}`.

Table 1: The preamble options.

- The <–option was 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.

1.1 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 centred LR-mode in an `array`, or centred math-mode in a `tabular`.

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

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

The above solution always centres 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.

[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.

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

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.

1.2 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.

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 Comparisons with older versions of `array.sty`

There are some differences in the way version 2.1 treat 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]` is 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 did generate an error in earlier versions, but are new 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).

2.2 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 centred 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$\bf$#$ $\hfil` and so the column entry would not be in the scope of the `\bfseries`! The present version switches the order of repeated declarations, and so the above example now produces a preamble of the form `\hfil$ $\bfseries#$ $\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 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⟩
\documentclass{ltxdoc}

\AtBeginDocument{\DeleteShortVerb{\|}}  % undo the default is not used

 \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⟩
```

4 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.

`\@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 `latex.tex`, which is the reason why the following three lines of code are commented out. Later throughout the text I will not mention it again explicitely whenever I use a `%` sign. These parts are already defined in `latex.tex`.

```
% \newcount \@chclass
```

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

Table 2: Classes of preamble tokens

```
% \newcount \@chnum
% \newcount \@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}}
```

4.1 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`.

```
\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}
```

4.2 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.

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

`\@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.

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

5 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 9). The advantage of our approach is the fact that we avoid some of the problems that arise with the other method[3].

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

[3]Maybe there are also historical reasons.

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.

`\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@{}}
```

`\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 12). 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 `\relex` 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.

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 formating 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

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

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

Here, we assume that the count register \@tempcnta has saved the value \count@ − 1.

```
    \the@toks \the \@tempcnta
```

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 text, we set an \ignorespaces in front and a \unskip afterwards.

```
    \ignorespaces \@sharp \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}
```

5.1 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 \@addtopreamble &\fi}
```

\@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}
```

5.2 The macro \@mkpream

\@mkpream \the@toks

Now we can define the macro which builds up the preamble for the \halign. First we 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
```

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}}
```

6 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}
```

`\@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 determinated 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
```

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`.

```
     \d@llarend \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`.

```
      \d@llarbegin \insert@column \d@llarend \hfil \or
      \hfil\kern\z@ \d@llarbegin \insert@column \d@llarend \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 `$`.

```
   $\vcenter
```

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`.

```
   \@startpbox{\@nextchar}\insert@column \@endpbox $\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`.

```
   \vtop \@startpbox{\@nextchar}\insert@column \@endpbox \or
   \vbox \@startpbox{\@nextchar}\insert@column \@endpbox
```

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.

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

`\@classix` 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}
```

`\@classviii` 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}
```

`\@arrayrule` There is only one incompatibility with the original definition: the definition of `\@arrayrule`. In the original a line without width[4] 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}
```

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

\@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}
```

\@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}
```

\@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}
```

\@classv 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}
```

\@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}
```

\@startpbox 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
```

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

```
  \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{}}%
   }
```

\@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 \egroup}
```

7 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@}%
```

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}%
```

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.

```
\xdef\@preamble{\ialign \@halignto
               \bgroup \@arstrut \@preamble
                       \tabskip \z@ \cr}%
```

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*⟩. 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 12). 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. The PLAIN TeX–macro `\m@th` does this.

```
  \lineskip \z@
  \baselineskip \z@
  \m@th
```

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\\\let\par\@empty \@preamble}
```

`\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
```

`\@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}
```

8 The line separator \\

`\@arraycr` In the macro `\@array` the line separator `\\` is `\let` to the command `\@arraycr`. Its definition starts with a special brace which I have directly copied from the original definition. It is necessary, because the `\futurlet` 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 beused 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 realy produce a group at all but modifies the master counter so that a `&` will not be considered belonging to the current `\haling` while we are looking for a `*` or `[`. For further information see [2, Appendix D].

```
\def\@arraycr{\relax\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}
```

`\@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}}
```

`\@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 a depth of `\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}}
```

`\@xargarraycr`
`\@yargarraycr`

The following macros were already explained above.

```
\def\@xargarraycr#1{\unskip
  \@tempdima #1\advance\@tempdima \dp\@arstrutbox
  \vrule \@depth\@tempdima \@width\z@ \cr}
\def\@yargarraycr#1{\cr\noalign{\vskip #1}}
```

9 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.

```
\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
```

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
```

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}
```

10 The Environment Definitions

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

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

In order to relieve the save stack we assign the replacement texts for `\@halignto` globally. `\d@llar` has to be local since otherwise nested tabular and array environments (via `\multicolumn`) are impossible. 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
```

`\array`

Our new definition of `\array` then reads:

```
\def\array{\col@sep\arraycolsep
   \def\d@llarbegin{$}\let\d@llarend\d@llarbegin\gdef\@halignto{}%
```

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

```
   \@tabarray}
```

`\@tabarray`

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

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

`\tabular` `\tabular*`

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

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

and analoguesly

```
\expandafter\def\csname tabular*\endcsname#1{%
      \gdef\@halignto{to#1}\@tabular}
```

`\@tabular`

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

```
\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
```

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 $\col@sep\tabcolsep \let\d@llarbegin\begingroup
                                   \let\d@llarend\endgroup
```

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

```
\@tabarray}
```

`\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.

```
\def\endarray{\crcr \egroup \egroup \gdef\@preamble{}}
```

`\endtabular` `\endtabular*` To end a tabular or tabular* environment we call up `\endarray`, close the math mode and then the surrounding `\hbox`.

```
\def\endtabular{\endarray $\egroup}
\expandafter\let\csname endtabular*\endcsname=\endtabular
```

11 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.

```
\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
```

12 Defining your own column specifiers[5]

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

[5]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.

To allow processing older sources that made use of `newarray.sty` we provide a definition for the old name. However for LaTeX 2_ε we make it an error message.

```
⟨*ncols⟩
```

`\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{\if\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 occurences 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]}}
```

`\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}}
```

`\NC@` Having found an occurence 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}
```

`\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}
```

\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 initialises 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}
```

\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 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 itsef to print the rest of the definitions.

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

\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\@@}}
```

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

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

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

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

12.1 The ∗–form

We view the ∗-form as a slight generalisation 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
```

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}
```

12.2 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
```

\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@}}
```

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

```
\def\@mkpream#1{\gdef\@preamble{}\@lastchclass 4 \@firstamptrue
   \let\@sharp\relax \let\@startpbox\relax \let\@endpbox\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}}
```

`\@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
```

`\@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}
```

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

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

12.3 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]}}
```

`\@@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
```

`\endarray` `\@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.

```
\def\endarray{\crcr \egroup \egroup \@arrayright \gdef\@preamble{}}
\let\@arrayleft\@empty
\let\@arrayright\@empty
```

12.4 Support for `\firsthline` and `\lasthline`

The Companion [1, p.137] suggests two additional commands to control the allignments 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}
```

`\firsthline` `\lasthline` `\backup@length` This code can probably be improved but for the moment it should serve.

```
\newlength{\backup@length}
\newcommand{\firsthline}{%
  \multicolumn1c{%
    \global\backup@length\ht\@arstrutbox
    \global\advance\backup@length\dp\@arstrutbox
```

```
% Done locally to be restored to correct value for backup in \\
    \advance\backup@length\extratabsurround
% This gives some extratab space above the line
    \rule\z@\backup@length
    }\\[-\backup@length]\hline
}
% This provides the same space at the bottom
\newcommand{\lasthline}{\hline\multicolumn1c{}%
    \global\backup@length-\ht\@arstrutbox
    \global\advance\backup@length\extratabsurround
    \\[\backup@length]%
}
⟨/package⟩
```

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

\@@ 56, 67, 68, 84, 262, 263
\@@@ 307, 310
\@@array . . 314, 315, 315
\@@endpbox 224
\@acol . . . 79, 80, 81, 97, 99, 108, 111, 138, 140, 295, 297
\@acolampacol 79, 81, 107, 109
\@addamp 76, 77, 81, 108, 112, 195
\@addtopreamble 30, 30, 78, 80, 101, 117, 134, 139, 146, 197, 299, 308
\@ampacol 220
\@argarraycr 185, 186, 186
\@argtabularcr 225
\@array 158, 158, 208, 315
\@arrayacol 223
\@arrayclassiv 221
\@arrayclassz 221
\@arraycr . 178, 182, 182
\@arrayleft . . . 171, 317
\@arrayparboxrestore 152
\@arrayright 316, 316, 318
\@arrayrule 134, 134, 149
\@arstrut 168, 181, 181, 200
\@arstrutbox . . 154, 157, 161, 181, 190, 324, 325, 334
\@chclass 27, 27, 31, 43, 54, 91, 95, 133, 288, 293, 305
\@chnum 27, 28, 32, 39, 50, 117, 149
\@classi 91, 148, 148, 288
\@classii 91, 143, 143, 288
\@classiii 143
\@classix 93, 129, 129, 303, 303
\@classv 92, 145, 145, 150, 289, 306, 306
\@classvi 92, 133, 137, 137, 148, 289, 305
\@classvii 93, 135, 135, 141, 290
\@classviii . 93, 132, 132, 290, 304, 304
\@classx 94, 105, 105, 114, 131, 291, 292
\@classz 91, 114, 114, 288
\@empty 178, 197, 311, 317, 318
\@endpbox . 83, 125–127, 157, 157, 279
\@expast 220
\@finalstrut 157
\@firstampfalse 77, 111, 195
\@firstamptrue . 82, 278
\@halignto . . . 167, 203, 206, 209, 211
\@iffirstamp 76
\@lastchclass 27, 29, 32–35, 37, 82, 95, 96, 106, 129, 132, 135, 137, 278, 293, 294, 304
\@mkpream . . . 82, 82, 166, 197, 278, 278
\@namedef 243, 266
\@nextchar 38, 40–42, 44–48, 51–53, 71, 88, 125–127, 228, 277, 285, 307
\@preamble . . 30, 82, 167, 168, 178, 200, 217, 278, 316
\@preamerr 54, 100, 101, 103, 130, 133, 136, 196, 226, 226, 298, 299, 301, 305
\@reargdef 245
\@sharp 72, 74, 83, 101, 174, 199, 279, 299
\@startpbox 83, 125–127, 151, 151, 279
\@tabacol 223
\@tabarray 207, 208, 208, 216, 313, 314
\@tabclassiv 222

\@tabclassz 222
\@tabular 209, 211, 212, 212
\@tabularcr 224
\@tempswafalse 281
\@tempswatrue . 251, 280
\@temptokena 58, 65, 248, 257, 258, 271, 280, 286
\@testpach 31, 31, 90, 287
\@tfor 88, 238, 285
\@whilesw 281
\@xargarraycr 187, 189, 189
\@xarraycr 183, 184, 184
\@xexnoop 64, 68, 68, 274, 275
\@xexpast 56, 56, 63, 84, 274, 274
\@xtabularcr 225
\@yargarraycr 188, 189, 192

A

\array 205, 205
\arraycolsep 205
\arraystretch . 162, 163
\AtBeginDocument . . . 4

B

\backup@length 321, 321, 324, 325, 327, 329, 330, 334–336
\begin 20
\bottomfraction 9

C

\CodelineIndex 16
\col@sep 79, 79, 80, 205, 214
\count@ . . 69, 69, 70, 71, 75, 85, 115, 143, 146, 267, 269, 270, 276, 277, 282, 308, 311

D

\d@llarbegin 119, 122, 123, 146, 203, 203, 206, 214, 308
\d@llarend . . . 121–123, 146, 203, 204, 206, 215, 308
\DeleteShortVerb . . . 4
\DisableCrossrefs . 12
\DocInput 21
\documentclass 2
\doublerulesep 139

E

\EnableCrossrefs . . 11
\end 22
\endarray 217, 217, 218, 316, 316
\endgroup 170, 198, 204, 215
\endtabular 218, 218, 219
\endtabular* 218
\everypar 153, 155
\extracolsep . . 307, 310
\extrarowheight . *1*, 160, 179, 179, 180
\extratabsurround *5*, 319, 319, 320, 327, 335

F

\firsthline . *4*, 321, 322
\futurelet 248

G

\global 324, 325, 334, 335

H

\hline 330, 333

I

\ialign 167
\if@firstamp . . . 77, 195
\if@tempswa 281
\iffalse 182
\insert@column 72, 72, 120, 122, 123, 125–127

L

\lasthline . *4*, 321, 333
\long 266
\loop 268

M

\m@th 177
\meaning 262
\multicolumn 193, 193, 323, 333

N

\NC@ 243, 247, 247
\NC@char 236–238, 240, 242–244
\NC@do 241, 252, 252, 259
\NC@ecs . . . 307, 310, 310
\NC@find 246, 255, 258, 273
\NC@ifend . 248, 249, 249
\NC@list 241, 259, 264, 264, 281
\NC@rewrite . . . 251, 253
\NC@rewrite@* 265
\NC@show . . 259, 260, 260
\NC@strip . 261, 263, 263
\NeedsTeXFormat . . . 25
\newcol@ . . 244, 245, 245
\newcolumn 234
\newcolumntype *3*, 235, 235, 265
\newcommand . . . 322, 333
\newif 76
\newlength . . . 319, 321
\newtoks 264
\next 238, 248, 250
\noexpand 238
\null 201

O

\OldMakeindex 19
\OnlyDescription . . 18

P

\PackageError 227
\PackageWarning 239, 242
\par 178
\prepnext@tok . . 69, 69, 87, 116, 128, 144, 147, 284, 309
\protect 174
\ProvidesPackage . . 26

R

\RecordChanges 14
\renewcommand 9
\repeat 272
\rule 329

S

\save@decl 71, 71, 92, 144, 145, 276, 276, 289, 306
\setlength 320
\showcols . . *4*, 259, 259
\string 38, 236, 254, 256

T

\tabcolsep 214

\tabular 209, 209
\tabular* 209
\tabularnewline . . . 178
\the@toks 56, 59, 62, 65, 73, 75, 82, 86, 104, 146, 283, 302, 308
\the@toksz 56, 59, 60, 65

U

\usepackage 6

Change History

1994/12/08
  \@array: add \tabularnewline . 20

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

v1.9a
  "General": 2) 'protect is no longer necessary. But still the macro '@expast needs top be modified. 'multicolumn still does not work. 1
  Last (so I hope) major change: 1) Options B,A now called ¿,¡. These options now point to the column they modify. 1

v1.9b
  "General": inserted missing 'fi in '@testpach. Corrected LaTeXbug in '@tfor. 1

v1.9c
  "General": 1) 'def 'the@toks 'the ... remaining only in '@mkpream. 2) Removed '@classiii and replaced by 'save@decl. 1

3) ‘insert@column contains only ‘@tempcnta and ‘count@ counters. 4) ‘@@startpbox and ‘@@endpbox now totally obsolete. . . . 1
Re-introduced ‘@endpbox. ‘multicolumn now works Version number still 1.9 since the documentation is still not finished. 1

v1.9d
"General": Replaced ‘number by ‘the where the ‘toks registers' contents are used. . . . 1

v1.9e
"General": Re-introduced ‘@xargarraycr and ‘@yargarraycr, since ‘endtemplate seems to be ‘outer. . . . 1

v1.9f
"General": Small changes finally carried out: 1) ‘par=‘@empty. 2) ..ifnum0=‘... → ‘bgroup and analoguously ‘egroup. . . . 1

v1.9g
"General": Inserted again ..ifnum0=‘.., c.f. Appendix D of the TEXbook. . . . 1

v1.9h
"General": No longer necessary to read in the file twice. . . . 1

v1.9i
"General": Corrected typo in german version. . . . 1

v1.9j
"General": In a ‘r' column an extra ‘kern‘z@ is needed. . . . 1
Otherwise the ‘hfil on the left side will be removed by the ‘unskip in ‘insert@column if the entry is empty. . . . 1

v1.9k
"General": ‘beginMacro changed to ‘beginmacro in documentation. 1
Corrected typo in german version. . . . 1

v2.0a
\@testpach: p option renamed to m (middle). . . . 8
t option renamed to p to be compatible to the original. . . . 8
"General": \@thetoks changed to \the@toks. . . . 1
File renamed from arraye.sty to array.sty. . . . 1
source changed to reflect new doc.sty conventions. . . . 1
t option renamed to p to be compatible to the original. . . . 1

v2.0b
"General": All lines shortened to 72 or less. . . . 1
Three forgotten end macro added. . . . 1

v2.0c
\@classv: \relax added to avoid problem ‘the‘toks0‘the‘toks1. . 17
\@sharp: \relax added to avoid problem \the\toks0\the\toks1. . . . 12
\save@decl: \relax removed and added elsewhere. . . . 11

v2.0d
\@tabular: ‘d@llar local to preamble. . . . 23
\array: ‘d@llar local to preamble. 22

v2.0e
\@sharp: Added around \@sharp for new ftsel . . . 12

v2.0f
\@testpach: Argument removed since implicitly known . . . 8
Ensure to test a char which is not active . . . 8
\the@toks: \@testpach now without arg . . . 13

v2.0g
\d@llarend: ‘d@llarbegin defined on toplevel. . . . 22

v2.0h
\@sharp: Removed again in favour of \d@llarbegin . . . 12

v2.1a
\@array: Hook for delarray added 19
Wrong spec is now equiv to [t] 19
"General": Newcolumn stuff added 23

v2.1b
\newcolumntype: Macro renamed from ‘newcolumn . . . 24

v2.1c
\@startpbox: Use ‘everypar to insert strut . . . 18

v2.2a
"General": Upgrade to LATEX 2ε . . 1
\newcolumn: Now made ‘newcolumn an error . . . 24
Removed ‘newcolumn . . . 24

v2.2b

"General": Removed interactive prompt 6

v2.2c

"General": removed check for \@tfor bug 1

v2.2d

\@endpbox: Use LaTeX 2_ε \@finalstrut 18

v2.2e

\multicolumn: Added \null . . . 22

v2.3a

"General": Added code for \firsthline and friends 1

v2.3c

\@argarraycr: Avoid adding an ord atom in math 20

Use \expandafter's in conditional 21

\@arraycr: Avoid adding an ord atom in math 20

\@xarraycr: Avoid adding an ord atom in math 20

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.