A new implementation of \LaTeX’s \texttt{tabular} and \texttt{array} environment

Frank Mittelbach  David Carlisle

Printed September 23, 2022

Abstract

This article describes an extended implementation of the \LaTeX\ array– and \texttt{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 \texttt{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 \texttt{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.

Additionally we introduce a new parameter called \texttt{\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 \texttt{\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 \texttt{\bfseries}) in a flushed left column, this can be done with \texttt{>{\bfseries}l}. You do not have to begin every entry of the column with \texttt{\bfseries} any more.
<table>
<thead>
<tr>
<th>Unchanged options</th>
</tr>
</thead>
<tbody>
<tr>
<td>l</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>r</td>
</tr>
<tr>
<td>p{width}</td>
</tr>
<tr>
<td>@{decl.}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>New options</th>
</tr>
</thead>
<tbody>
<tr>
<td>m{width}</td>
</tr>
<tr>
<td>b{width}</td>
</tr>
<tr>
<td>){decl.}</td>
</tr>
<tr>
<td>){decl.}</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>){decl.}</td>
</tr>
<tr>
<td>w{align}{width}</td>
</tr>
<tr>
<td>W{align}{width}</td>
</tr>
</tbody>
</table>

Table 1: The preamble options.

- In columns which have been generated with \texttt{p}, \texttt{m} or \texttt{b}, the default value of \texttt{parindent} is \texttt{0pt}. This can be changed with \texttt{>{\setlength{\parindent}{1cm}}p}.
- The >– and <–options were originally developed for the following application: \texttt{>{\$}c<<{$}} generates a column in math mode in a \texttt{tabular}–environment. If you use this type of a preamble in an \texttt{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 \texttt{>{\centerdots}c <<{\endcenterdots}}. To center decimals at their decimal points you (only?)
have to define the following macros:

\begin{verbatim}
{\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}
\end{verbatim}

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 \texttt{dcolumn.sty} by David Carlisle.

- Using \texttt{c!{\hspace{1cm}}c} you get space between two columns which is enlarged by one centimeter, while \texttt{c@{\hspace{1cm}}c} gives you exactly one centimeter space between two columns.

- A declaration like \texttt{w{l}{3cm}} (or even shorter \texttt{wl{3cm}}) works like an \texttt{l} column except that the width will always be 3 cm regardless of the cell content. Same with \texttt{w{c}} or \texttt{w{r}}. This means that it is easy to set up tables in which all columns have predefined widths.

1.1 The behavior of the ``\ command

In the basic \texttt{tabular} implementation of \LaTeX{} the ``\bslash{} command ending the rows of the \texttt{tabular} or \texttt{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 \cite{lamport94} may not always produce the expected extra space.

Without the \texttt{array} package the extra space requested by the optional argument of ``\bslash{} 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{verbatim}
\begin{tabular}{lp{1cm}}
 1 & newline x \[20pt]\end{tabular}
\begin{tabular}{p{1cm}l}
 1 & newline 1 & x \[20pt]\end{tabular}
\end{verbatim}

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

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

\[
\begin{array}{c c c c}
1 & 1 & 1 & x \\
1 & x & 1 & \\
2 & 2 & & \\
\end{array}
\]

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):

\[
\begin{array}{c c c c}
1 & 1 & 1 & x \\
1 & x & 1 & \\
2 & 2 & 2 & 2 \\
\end{array}
\]

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.

### 1.2 Defining new column specifiers

Whilst it is handy to be able to type

```
\newcolumntype{⟨⟨some declarations⟩⟩}{⟨⟨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

\[
\text{\texttt{\newcolumntype{x}{{⟨⟨some declarations⟩⟩}{⟨⟨some more declarations⟩⟩}}}}
\]

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}{>{⟨⟨some declarations⟩⟩}{⟨⟨some more declarations⟩⟩}c<⟨⟨some declarations⟩⟩{⟨⟨some more declarations⟩⟩}}
\newcolumntype{L}{>{⟨⟨some declarations⟩⟩}{⟨⟨some more declarations⟩⟩}l<⟨⟨some declarations⟩⟩{⟨⟨some more declarations⟩⟩}}
\newcolumntype{R}{>{⟨⟨some declarations⟩⟩}{⟨⟨some more declarations⟩⟩}r<⟨⟨some declarations⟩⟩{⟨⟨some more declarations⟩⟩}}
```

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

\[^{1}\text{This command was named } \texttt{\textbackslash newcolumn} \text{ in the } \texttt{newarray.sty}. \text{ At the moment } \texttt{\textbackslash newcolumn} \text{ is still supported (but gives a warning). In later releases it will vanish.}\]
where the appropriate macros in this case are:\(^2\)

\begin{verbatim}
def\coldot{}% Or if you prefer, \def\coldot{\cdot}
{\catcode'.=\active
 gdef.{$\egroup\setbox2=\hbox to \dimen0 \bgroup$\coldot})
def\rightdots#1{% \setbox0=\hbox{$\coldot$}\advance\dimen0 \wd0 \setbox0=\hbox{$\coldot$}\advance\dimen0 \wd0 \setbox2=\hbox to \dimen0 {}}
def\endrightdots{$\hfil\egroup\box0\box2}
\end{verbatim}

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

A rather different use of the \texttt{\newcolumntype} system takes advantage of the fact that the replacement text in the \texttt{\newcolumntype} command may refer to more than one column. Suppose that a document contains a lot of \texttt{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).

\begin{verbatim}
\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\}
\end{verbatim}

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

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

### 1.3 Special variations of \texttt{\hline}

The family of \texttt{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 \texttt{t} or \texttt{b} value to the optional position argument. However, this does not work when the first or last element in the environment is a \texttt{\hline} command—in that case the environment is aligned at the horizontal rule.

\(^2\)The package \texttt{dcolumn.sty} contains more robust macros based on these ideas.
Here is an example:

\begin{tabular}{|l|}
\hline
with some hline commands used
\end{tabular}

\begin{tabular}{|l|}
\hline
with some hline commands used
\end{tabular}

Using \texttt{\firsthline} and \texttt{\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.

The implementation of these two commands contains an extra dimension, which is called \texttt{\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 \texttt{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 \texttt{lll} to \texttt{l|l|l} does not affect the document other than adding rules to the table. In contrast, with \texttt{array.sty} a table that just fit the \texttt{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 $\texttt{\tabcolsep}$, 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 \texttt{array.sty} modifying any such length usually works well as the actual visual white space (from $\texttt{\tabcolsep}$, 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 $\texttt{\arrayrulewidth}$ is chosen. In that case a simple table like

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

will produce something like

\begin{table}
\centering
\begin{tabular}{|l|}
\hline
A \\
\hline
\end{tabular}
\end{table}

Horizontal rules produced with $\texttt{\hline}$ add to the table height in both implementations but they differ in handling double $\texttt{\hline}$s. In contrast a $\texttt{\cline}$ does not change the table height.$^3$

2.2 Comparisons with older versions of \texttt{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 \texttt{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 $\texttt{\extracolsep}$ command does not work with the old versions of \texttt{array.sty}, see the comments in \texttt{array.bug}. With version 2.1 $\texttt{\extracolsep}$ may again be used in $\Phi$-expressions as in standard \LaTeX{}, and also in $!$-expressions (but see the note below).

\footnote{\textsuperscript{3}All a bit inconsistent, but nothing that can be changed after being 30+ years in existence.}
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.

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 `>`{\(\text{decs1}\)}{\(\text{decs2}\)}`}`{\(\text{decs1}\)}{\(\text{decs2}\)`} the same as `>`{\(\text{decs1}\)}{\(\text{decs2}\)}`}. 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 `>`{\(\text{bfseries}\)C}, which re-writes to `>`{\(\text{bfseries}\)C}{\$}\(c\){\$} would not specify a bold column as might be expected, as the preamble would essentially expand to `\hfil$\text{bfseries}$$\text{bfseries}\hfil$ and so the column entry would not be in the scope of the `\text{bfseries}`! The present version switches the order of repeated declarations, and so the above example now produces a preamble of the form `\hfil$\text{bfseries}$$\text{bfseries}\hfil$, and the dollars cancel each other out without limiting the scope of the `\text{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}}\ast\ef\ast` does not work with this package. However you can use something like `\newcolumntype{e}@{\extracolsep{\fill}}` instead.
- As noted by the L\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|` 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
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

\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\{\hskip1cm\}\} consists of the token l, c, l, l, \hskip and \hskip1cm.

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
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.

<table>
<thead>
<tr>
<th>token</th>
<th>@chclass</th>
<th>@chnum</th>
<th>token</th>
<th>@chclass</th>
<th>@chnum</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>Start</td>
<td>4</td>
<td>—</td>
</tr>
<tr>
<td>l</td>
<td>0</td>
<td>1</td>
<td>@-arg</td>
<td>5</td>
<td>—</td>
</tr>
<tr>
<td>r</td>
<td>0</td>
<td>2</td>
<td>!</td>
<td>6</td>
<td>—</td>
</tr>
<tr>
<td>m-arg</td>
<td>0</td>
<td>3</td>
<td>@</td>
<td>7</td>
<td>—</td>
</tr>
<tr>
<td>p-arg</td>
<td>0</td>
<td>4</td>
<td>&lt;</td>
<td>8</td>
<td>—</td>
</tr>
<tr>
<td>b-arg</td>
<td>0</td>
<td>5</td>
<td>&gt;</td>
<td>9</td>
<td>—</td>
</tr>
<tr>
<td>l</td>
<td>1</td>
<td>0</td>
<td>m</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>!-arg</td>
<td>1</td>
<td>1</td>
<td>p</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>&lt;-arg</td>
<td>2</td>
<td>—</td>
<td>b</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>&gt;-arg</td>
<td>3</td>
<td>—</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Classes of preamble tokens

\@chclass The class and the number of the current token are saved in the count registers \@chnum 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.

\@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. We will not discuss our example further but rather take a look at the general case of constructing preambles.

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
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 \one \@chnum \one \else
\ifnum \@lastchclass=7 5 \else
\ifnum \@lastchclass=8 \two \else
\ifnum \@lastchclass=9 \three

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

\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 ~ \else
\if \@nextchar ! \six \else
\if \@nextchar @ \seven \else
\if \@nextchar < \eight \else
\if \@nextchar > \nine \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.

\if \@nextchar | \one \else
\if \@nextchar !\six \else
\if \@nextchar @\seven \else
\if \@nextchar <\eight \else
\if \@nextchar >\nine \else

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

\if \@nextchar m\three \else
\if \@nextchar p\four \else
\if \@nextchar b\five \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.
4.2 Multiple columns (∗-form)

\@xexpast Now we discuss the macro that deletes all forms of type \(\{N\}\{\text{String}\}\) from a
\the@toks user preamble and replaces them with \(N\) copies of \text{String}. Nested \(\ast\)-expressions
\the@toksz are dealt with correctly, that means \(\ast\)-expressions are not substituted if they are
in explicit braces, as in \(\{\ast\}\).

This macro is called via \(\@xexpast\langle\text{preamble}\rangle\ast0x\@@\). The \(\ast\)-expression \(\ast0x\) 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 \(\ast\)-expression while the second and third ones are the arguments of the first \(\ast\)-expression (that is \(N\) and \text{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 \text{String} (\#2) that are to be produced will be saved in a count register.

\let\@tempcnta\@tempcnta\% We save the part of the preamble which does not contain a \(\ast\)-form (\#1) in a
Plain \TeX{} token register. We also save \text{String} (\#3) using a LR\TeX{} token register.

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

Now we have to use a little trick to produce \(N\) copies of \text{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 \(\{\ldots\}\) 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\% \edef\@tempa{\@tempa\the@toks}\%

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 appened \(\ast0x\@@\) when we first called \@xexpast. In other words: if the user inserts
\(\{0\ldots\}\) in his preamble, LR\TeX{} 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\@tempa{\the@toksz\def\@tempa{\the@toks\% \def\@tempa{\the@toks\%}

We have now evaluated the first \(\ast\)-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 \(\ast\)-expression is handled, or to
the macro \@xexnoop, which only ends the recursion by deleting its argument.

\expandafter \@tempb \@tempa \#1\@@}
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.

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

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

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

We will solve this problem with \texttt{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 \texttt{token} registers. But then, who needs tables with more than 100 columns?

One could also find a solution which only needs two or three \texttt{token} registers by proceeding similarly as in the macro \@xexpast (see page 12). The advantage of our approach is the fact that we avoid some of the problems that arise with the other method\footnote{Maybe there are also historical reasons.}.

So how do we proceed? Let us assume that we had !\{foo\} in the user preamble and say we saved foo in \texttt{token} register 5. Then we call \@addtopreamble{\the@toks5} where \texttt{\the@toks} is defined in a way that it does not expand (for example it could be equivalent to \texttt{\relax}). Every following call of \@addtopreamble leaves \texttt{\the@toks} unchanged in \@preamble. If the construction of the preamble is completed we change the definition of \texttt{\the@toks} to \texttt{\the\toks} and expand \@preamble for the last time. During this process all parts of the form \texttt{\the@toks(Number)} will be substituted by the contents of the respective \texttt{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 \texttt{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 \texttt{count} register to remember which \texttt{token} register is to be used next. This will be initialized with −1 if we want to begin with the \texttt{token} register 0. We use the Plain \TeX scratch register \texttt{\count@} because everything takes place locally. All we have to do is insert \texttt{\the@toks \the \count@} into the preamble. \texttt{\the@toks} will remain unchanged and \texttt{\the\count@} expands into the saved number.

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

\@prepnext@tok\{\advance \count@ \@ne
\toks\count@{}\}

Then we locally delete any contents the \texttt{token} register might have.

\save@decl During the construction of the preamble the current \texttt{token} is always saved in the macro \@nextchar (see the definition of \@mkpream on page 15). The macro \@save@decl saves it into the next free \texttt{token} register, i.e. in \texttt{\toks\count@}.
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 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

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

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.

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.

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 &.
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

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 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.

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.

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.

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.

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

96 \% \expandafter \@tfor \expandafter \@nextchar
97 \% \expandafter \expandafter \expandafter \expandafter :\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.

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

99 \% \ifcase \@chclass \@classz \or \@classi \or \@classii
100 \% \or \save@decl \or \or \@classv \or \@classvi
101 \% \or \@classvii \or \@classviii \or \@classix
102 \% \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.

103 \% \@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.

104 \% \ifcase\@lastchclass
If the last class equals 0 we add a \hskip \col@sep.
105 \% \@acol \or
If it equals 1 we do not add any additional space so that the horizontal lines do not exceed the vertical ones.

106 \% \or
Class 2 is treated like class 0 because a <{...} can only directly follow after class 0.

107 \% \@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 {...}.

108 \% \@preamerr \thr@@ \or
If \@lastchclass is 4 the user preamble has been empty. To continue, we insert a # in the preamble.

109 \% \@preamerr \tw@ \addtopreamble\@sharp \or
Class 5 is allowed again. In this case (the user preamble ends with @{...}) we need not do anything.

110 \% \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{\% \ifcase \@lastchclass

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 \@firststampfalse. This makes sure that a later \@addamp inserts the character & into the preamble.

\% \@acol \@firststampfalse \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 \ldots\!p\ldots\!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:

122 \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.

123 \@tempcnta \count@

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

124 \prepnext@tok

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

125 \@addtopreamble{\ifcase \@chnum

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

126 \hfil

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

127 \hskip1sp%

The command \dollarbegin follows expanding into \begingroup (in the tabular–environment) or into \$. Doing this (provided an appropriate setting of \dollarbegin) 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.

128 \dollarbegin

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.

129 \insert@column

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

130 \dollarend \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.

131 \hskip1sp\dollarbegin \insert@column \dollarend \do@row@strut \hfil \or

132 \hfil\hskip1sp\dollarbegin \insert@column \dollarend \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 \$.

133 \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. The strut is placed after the box

\@startpbox{\@nextchar}\insert@column \@endpbox
\ar@align@mcell
\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.
\vtop \@startpbox{\@nextchar}\insert@column \@endpbox\do@row@strut \or
\vbox \@startpbox{\@nextchar}\insert@column \@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.
\fi\prepnext@tok

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

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

\begin{verbatim}
\def\@@classix{\ifnum \@lastchclass = \thr@@ \else \fi}
\end{verbatim}

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

\@@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 \texttt{token} is a \texttt{\textless} 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 \texttt{\textless} is treated
like \texttt{!}.

\begin{verbatim}
\def\@@classviii{\ifnum \@lastchclass >\z@ \else \fi}
\end{verbatim}

\arrayrule  There is only one incompatibility with the original definition: the definition of
\texttt{\arrayrule}. In the original a line without width\footnote{So the space between \texttt{cc} and \texttt{c|c} is equal.} is created by multiple in-
sertions of \texttt{\hskip .5\arrayrulewidth}. We only insert a vertical line into the
preamble. This is done to prevent problems with \TeX\’s main memory when gen-
erating tables with many vertical lines in them (especially in the case of \texttt{floats}).

\begin{verbatim}
\def\arrayrule{\@addtopreamble \vline}
\end{verbatim}

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

\begin{verbatim}
\def\@@classvii{\ifnum \@lastchclass = \thr@@ \else \fi}
\end{verbatim}

If this is true we output an error message and ignore the declarations stored by
the last \texttt{\ldots}, because these are overwritten by the argument of \texttt{\textatmark}.

\@@classvi  If the current \texttt{token} is a regular \texttt{!} and the last class was 0 or 2 we extend the
preamble with \texttt{\hskip \col@sep}. If the last \texttt{token} was of class 1 (for instance \texttt{!}) we extend with \texttt{\hskip \doublerulesep} because the construction \texttt{\textatmark\ldots} has to
be treated like \texttt{!}.
\def\@classvi{\ifcase \@lastchclass
\@acol \or
\@addtopreamble{\@skip \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
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 \dollar...’s.
% \@addtopreamble{\dollarbegin\the@toks\the\count\relax\dollarend}\%
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

21
or (in case of \{\ldots\}) the current token — stored in \texttt{@nextchar} — has to be inserted into the preamble. This corresponds to calling \texttt{@classv}.

\begin{verbatim}
\@classv \fi}
\end{verbatim}

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

\begin{verbatim}
\def\@startpbox#1{\bgroup
\color@begingroup
The argument is the width of the box. This information has to be assigned to \texttt{hsize}. Then we assign default values to several parameters used in a \texttt{parbox}.

\setlength{\hsize}{#1}\@arrayparboxrestore
\end{verbatim}

Our main problem is to obtain the same distance between succeeding lines of the \texttt{parbox}. We have to remember that the distance between two \texttt{parboxes} should be defined by \texttt{@arstrut}. That means that it can be greater than the distance in a \texttt{parbox}. Therefore it is not enough to set a \texttt{@arstrut} at the beginning and at the end of the \texttt{parbox}. This would dimension the distance between first and second line and the distance between the two last lines of the \texttt{parbox} wrongly. To prevent this we set an invisible rule of height \texttt{@arstrutbox} at the beginning of the \texttt{parbox}. This has no effect on the depth of the first line. At the end of the \texttt{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 \texttt{parindent} changes via \texttt{\{\ldots\}}.

\begin{verbatim}
\everypar{\vrule \@height \ht\@arstrutbox \@width \z@}
\everypar{}}\%}
\end{verbatim}

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

\begin{verbatim}
\def\@endpbox{\@finalstrut\@arstrutbox \par \color@endgroup \egroup\hfil}
\end{verbatim}

7 Building and calling \texttt{\halign}

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

\begin{verbatim}
\def\@array[#1]{\%}
\end{verbatim}

First we define a \texttt{strut} whose size basically corresponds to a normal \texttt{strut} multiplied by the factor \texttt{@arraysstretch}. This \texttt{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 \texttt{strut} the value of the parameter \texttt{@extrarowheight}.

\begin{verbatim}
\@tempdima \ht \strutbox
\end{verbatim}
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.

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 it 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).

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.

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.

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.)

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 15). All these now get their actual meaning.

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 \texttt{array}-environment the parameter \texttt{\mathsurround} should also be set to 0pt. This prevents additional space between the rows. The PLAIN \TeX-macro \texttt{\mathstrut} does this.

199 \lineskip \z@ 200 \baselineskip \z@ 201 \mathstrut

Beside, we have to assign a special meaning (which we still have to specify) to the line separator \texttt{\textbackslash}. We also have to redefine the command \texttt{\par} in such a way that empty lines in \texttt{\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 \texttt{\@preamble} to start the wanted \texttt{\halign}.

\arraybackslash

\texttt{\arraybackslash} Restore \texttt{\textbackslash} for use in array and tabular environment (after \texttt{\raggedright} etc.).

202 \let\textbackslash@arraycr \let\textbackslash\tabularnewline\let\par\@empty \@preamble}

\extrarowheight

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

203 \let\extrarowheight=0pt \@arstrut

\texttt{\@arstrut} Now the insertion of \texttt{\@arstrutbox} through \texttt{\@arstrut} is easy since we know exactly in which mode \LaTeX{} is while working on the \texttt{\halign} preamble.

204 \let\extrarowheight=0pt \@arstrut

8 The line separator \texttt{\textbackslash}

\@arraycr

In the macro \texttt{\array} the line separator \texttt{\textbackslash} is \texttt{\let} to the command \texttt{\@arraycr}. Its definition starts with a special brace which I have directly copied from the original definition. It is necessary, because the \texttt{\futurlet} in \texttt{\@ifnextchar} might expand a following \& token in a construction like \texttt{\textbackslash \&}. 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. \texttt{{\textbackslash}}, because the command will also be used for the array environment, i.e. in math mode. In that case an extra \texttt{{\textbackslash}} 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 \texttt{\halign} while we are looking for a * or [. For further information see \cite[Appendix D]{article}.

205 \protected\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).

206 \@ifstar \@xarraycr \@xarraycr}

\@xarraycr

In the command \texttt{\@xarraycr} we test if an optional argument exists.

207 \@ifnextchar \@xarraycr \@xarraycr \%}

If it does, we branch out into the macro \texttt{\@xarraycr} if not we close the special brace (mentioned above) and end the row of the \texttt{\halign} with a \texttt{\cr}.
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. If this is the case we create an invisible vertical rule with depth `\dp\@arstutbox + \langle wanted space \rangle`. 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.

The following macros were already explained above.

\argarraycr The following macros were already explained above.
\yargarraycr
\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.

\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.

9 Spanning several columns
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 \def\@preamble{\@preamble}. This is achieved shorter by writing:
\begin{verbatim}
\@mkpream(#2)\@addtopreamble\@empty
\end{verbatim}
After the \@preamble is created we forget all local definitions and occupations of the token registers.
\begin{verbatim}
\endgroup
\end{verbatim}
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.
\begin{verbatim}
\def\@sharp{#3}\
\end{verbatim}
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 \hspace{1em}) to remove the \tabcolsep.
\begin{verbatim}
\@arstrut \@preamble
\null
\ignorespaces}
\end{verbatim}
10 The Environment Definitions

After these preparations we are able to define the environments. They only differ in the initialisations of \dollar\ldots, \col_sep and \@halignto.
\begin{verbatim}
\@halignto \dollar has to be locally asigned 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 \halignto entrys 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 \dollarbegin and \dollarend between groups and dollar signs.
\let\dollarbegin\begingroup
\let\dollarend\endgroup
\array Our new definition of \array then reads:
\begin{verbatim}
\def\array{\col_sep\arraycolsep
\def\dollarbegin{$}\let\dollarend\dollarbegin\def\@halignto{\%
Since there might be an optional argument we call another macro which is also used by the other environments.
\@tabarray}
\@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.
\end{verbatim}
This macro tests for an 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 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 \texttt{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}

\@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 \dollar....

\hbox \bgroup \$\col@sep\tabcolsep \let\dollarbegin\begingroup
\let\dollarend\endgroup

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

\@tabarray}

\endarray

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.

When the processing of \texttt{array} is finished we have to close the \@align 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. 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}
\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

27
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{\ldots} 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

\section{Defining your own column specifiers\footnote{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.}}

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

\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.

\begin{verbatim}
\def\newcolumntype#1{\NC@char
\edef\NC@char{\string#1}%
\@ifundefined{NC@find@\NC@char}%%
\@tfor
ext:=<>clrmbp@!|do
\if\expandafter\noexpand\ext\NC@char\PackageWarning{array}{Redefining primitive column \NC@char}\fi
\NC@list\expandafter{\the\NC@list\NC@do#1}}%
\end{verbatim}

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

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

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

```latex
\@ifnextchar[\{\newcol@\@char\}[0]\}
```

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.

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

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.

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

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

```latex
\def\NC@ifend{\ifx\next\relax\else\if@tempswa\expandafter\NC@rewrite\fi}\}
```

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 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.

```latex
\def\NC@do#1{\let\NC@rewrite\NC@rewrite@C \let\NC@find\NC@find@C}
```

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.

```latex
\expandafter[\@temptokena][\expandafter]\}
```

\NC@do
The 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 re-deﬁne \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 re-deﬁne itself to print the rest of the deﬁnitions.

\showcols 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.

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

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

\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}{
  \count@#1\relax
  \loop
  \ifnum\count@>0
    \advance\count@ by -1
    \#2*{\count@}{#2}
  \fi
}

Store the number.

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

\loop
\ifnum\count@>0
  \advance\count@ by -1
\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 These macros are used to expand \*-forms in array.sty. \let them to \relax to save space.

306 \let\@xexpast\relax
307 \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.

308 \def\save@decl\{\toks \count@ = \expandafter\expandafter\expandafter \{\expandafter\@nextchar\the\toks\count@\}
309 \@mkpream The main modification to \@mkpream is to replace the call to \@xexpast (which expanded \*-forms) by a loop which expands all $\newcolumntype$ specifiers.

310 \def\@mkpream#1{\gdef\@preamble{}\@lastclass 4 \@firstamptrue 
311 \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 is necessary to expand the \#1 at least once (since it will get \@nextchar as its value and need its content!)

312 \def\@startpbox##1{\unexpanded\expandafter{\expandafter \@startpbox\expandafter{##1}}}
313 \let\@endpbox\relax
314 \let\do@row@strut\relax
315 \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 $\toks0$.

316 $\@temptokena\{#1\}@tempswatrue
317 $\@whileswi{if@tempswa}{fi}{@tempswafalse\the\NC@list}$$\%$
Afterwards we initialize all registers and macros, that we need for the build-up of the preamble.

318 $\count0\m@ne$
319 \let\the@toks\relax
320 \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 $\@for$ to inspect each token in turn.

321 \expandafter \@for \expandafter \@nextchar
\@testpatch does not take an argument since array.sty 2.0h.

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

\@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.

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

\@addAllarray.sty explicitly allow class 2, as repeated < expressions are accepted by this package.

\let\@classix\relax
\@classviii

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

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).
\@@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.
\@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.
\endarray 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.
\backup@length This register will be used internally by \firsthline and \lasthline. 
\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.
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.
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 
And we should also cancel the guard otherwise we end up with two.
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.

\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%
}%
}

\section{12.5 Getting the spacing around rules right}

Beside a larger functionality \texttt{array.sty} has one important difference to the standard \texttt{tabular} and \texttt{array} environments: horizontal and vertical rules make a table larger or wider, e.g., \texttt{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}
\langle/\texttt{package}\rangle
12.6 Implementing column types \( w \) and \( \bar{W} \)

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

\begin{verbatim}
\ar@cellbox
\end{verbatim}

For \( w \) and \( \bar{W} \) column types we need a box to temporarily hold the cell content.

\begin{verbatim}
\newsavebox\ar@cellbox
\end{verbatim}

\begin{verbatim}
\newcolumntypew
\end{verbatim}

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.

\begin{verbatim}
\newcolumntype{w}[2]{%
\begin{lrbox}{\ar@cellbox}
\begin{dollarbegin}
\end{dollarbegin}
\frac{c}{\begin{lrbox}{\ar@cellbox}
\begin{dollarbegin}
\end{dollarbegin}\end{lrbox}}
\end{verbatim}

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

\begin{verbatim}
\let\hss\hfil
\end{verbatim}

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 \texttt{\multicolumn} rows overwriting the setting in which case \( c \) seems to be slightly better.

\begin{verbatim}
\end{verbatim}

At the end of the cell we end the \texttt{lrbox} environment so that all of the cell content is now in box \texttt{\ar@cellbox}. As a final step we put that box into a \texttt{\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.

\begin{verbatim}
\end{verbatim}

\begin{verbatim}
\newcolumntype{W}[2]{%}
\begin{lrbox}{\ar@cellbox}
\begin{dollarbegin}
\end{dollarbegin}
\frac{c}{\begin{lrbox}{\ar@cellbox}
\begin{dollarbegin}
\end{dollarbegin}\end{lrbox}}
\end{verbatim}

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

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.

Symbols
\begin{verbatim}
\texttt{\@@@} ........... 343, 346 \texttt{\@array} ........ 350, 351
\end{verbatim}

62, 73, 74, 92, 294, 295
Change History

1994/12/08 v1.0b
\@array: add \tabularnewline . 24 General: `@classi (faster), `@classvi
A in preamble means \& \& in \texttt{`halign}. ........................ 1

\textbf{v1.1a}  
General: New concept: \texttt{preamblechar}:  
c,l,r,C,L,R,A,p,t,\{,\}@. ........................ 1

\textbf{v1.1b}  
General: Again \( p \) like original \LaTeX{} and \( z \) for centered \texttt{`parbox}. ........................ 1

\textbf{v1.2a}  
General: Completely new implementation. ........................ 1

\textbf{v1.2b}  
General: \( I \) does no longer generate space at start or end of the preamble. Otherwise \texttt{`hline} is too long. ........................ 1
Enlarged \texttt{@arstrutbox} by 1pt (Test-Impl) with dimen \texttt{@strutheight}. ........................ 1

\textbf{v1.2c}  
General: Enlarged \texttt{@arstrutbox} by \texttt{`extrarowheight}. Thus you may avoid large characters to overprint a \texttt{`hline}. ........................ 1
Introduced \texttt{`m@th} in \texttt{@array} to allow non-zero values of \texttt{mathsurround}. ........................ 1
New dimen parameter \texttt{`extrarowheight} (default: 0pt). ........................ 1

\textbf{v1.2d}  
General: Completed the documentation. ........................ 1

\textbf{v1.2e}  
General: Bug fixed: \( A \) at start of preamble resulted in an error since \texttt{@mkpream} generated \texttt{@arstrut} \& \ldots{} as a preamble. ........................ 1

\textbf{v1.2f}  
General: Completed the documentation. ........................ 1

\textbf{v1.3a}  
General: Again a new implementation, with a new concept (cf. the documentation). ........................ 1

\textbf{v1.3b}  
General: \texttt{@decl} expands now into \texttt{@empty}, i.e., it disappears when the preamble is generated, except when the user specifies \( A \{ \) or \( B \{ \). ........................ 1

\textbf{v1.4a}  
General: Test implementation of use of token registers in order to do without \texttt{`protect}. ........................ 1

\textbf{v1.4b}  
General: Changed erroneous class numbers: 5 \( \rightarrow \) 6 6 \( \rightarrow \) 7 7 \( \rightarrow \) 5  
Corresponding changes in the macros. ........................ 1

\textbf{v1.4c}  
General: Everything except \( p,z \) now works with token registers. ........................ 1

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

\textbf{v1.9b}  
General: inserted missing \texttt{`fi} in \texttt{@testpach}. Corrected \LaTeX{} bug in \texttt{@@iffor}. ........................ 1

\textbf{v1.9c}  
General: 1) \texttt{`def} \texttt{`the}@toks \{\texttt{`the} \ldots{\}} \texttt{remaining only in \texttt{`mkpream}.}  
2) Removed \texttt{`classiii} and replaced by \texttt{`save@decl}. ........................ 1
3) \texttt{`insert@column} contains only \texttt{`tempcnta} and \texttt{`count@ counters}. 4) \texttt{`@startpbox} and \texttt{`@endpbox} now totally obsolete. ........................ 1
Re-introduced \texttt{`@endpbox}. \texttt{`multicolumn} now works!  
Version number still 1.9 since the documentation is still not finished. ........................ 1

\textbf{v1.9d}  
General: Replaced \texttt{`number} by \texttt{`the} where the \texttt{`toks registers} contents are used. ........................ 1

\textbf{v1.9e}  
General: Re-introduced \texttt{@xarrray} and \texttt{@yarray}, since \texttt{`endtemplate} seems to be \texttt{`outer}. ........................ 1

\textbf{v1.9f}  
General: Small changes finally carried out: 1) \texttt{`par} = \texttt{`empty}  
2) \texttt{\{\ldots{}ifnum0=} \ldots{\}} \rightarrow \texttt{`bgroupe} and analogously \texttt{`egroupe}. ........................ 1
<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1.9g</td>
<td>General: Inserted again ifnum=', c.f. Appendix D of the T\TeXbook</td>
</tr>
<tr>
<td>v1.9h</td>
<td>General: No longer necessary to read in the file twice.</td>
</tr>
<tr>
<td>v1.9i</td>
<td>General: Corrected typo in german version.</td>
</tr>
<tr>
<td>v1.9j</td>
<td>General: In a 'r' column an extra 'kern/\v0' is needed. Otherwise the 'hfil on the left side will be removed by the 'unskip in insert@column if the entry is empty.</td>
</tr>
<tr>
<td>v1.9k</td>
<td>General: 'beginMacro changed to 'beginmacro in documentation. Corrected typo in german version.</td>
</tr>
<tr>
<td>v2.0a</td>
<td>General: @thetoks changed to \the@toks. File renamed from arraye.sty to array.sty. Source changed to reflect new doc.sty conventions. t option renamed to p to be compatible to the original. @testpach: p option renamed to m (middle). t option renamed to p to be compatible to the original.</td>
</tr>
<tr>
<td>v2.0b</td>
<td>General: All lines shortened to 72 or less. Three forgotten end macro added.</td>
</tr>
<tr>
<td>v2.0c</td>
<td>\classv: \relax added to avoid problem 'the@toks0\the@toks1. @sharp: \relax added to avoid problem 'the@toks0\the@toks1. \save@decl: \relax removed and added elsewhere.</td>
</tr>
<tr>
<td>v2.0d</td>
<td>\tabular: 'dollar local to preamble. \array: 'dollar local to preamble.</td>
</tr>
<tr>
<td>v2.0e</td>
<td>@sharp: Added {} around \sharp for new ftsel</td>
</tr>
<tr>
<td>v2.0f</td>
<td>@testpach: Argument removed since implicitly known. Ensure to test a char which is not active. \the@toks: @testpach now without arg</td>
</tr>
<tr>
<td>v2.0g</td>
<td>\dollar: \dollarbegin defined on toplevel.</td>
</tr>
<tr>
<td>v2.0h</td>
<td>@sharp: Removed {} again in favour of \dollarbegin</td>
</tr>
<tr>
<td>v2.1a</td>
<td>General: Newcolumn stuff added. \array: Hook for delarray added. Wrong spec is now equiv to [t]</td>
</tr>
<tr>
<td>v2.1b</td>
<td>General: Removed interactive prompt.</td>
</tr>
<tr>
<td>v2.2a</td>
<td>General: Upgrade to L\A\TeX{} 2\epsilon. Newcolumn made 'newcolumn an error. Removed 'newcolumn</td>
</tr>
<tr>
<td>v2.2b</td>
<td>General: Removed interactive prompt.</td>
</tr>
<tr>
<td>v2.2c</td>
<td>General: removed check for @tfor bug.</td>
</tr>
<tr>
<td>v2.2d</td>
<td>@endpbox: Use L\A\TeX{} 2\epsilon@finalstrut</td>
</tr>
<tr>
<td>v2.2e</td>
<td>\multicolumn: Added \null</td>
</tr>
<tr>
<td>v2.3a</td>
<td>General: Added code for \firstline and friends</td>
</tr>
<tr>
<td>v2.3c</td>
<td>General: (DPC) minor doc changes. \argarrayc: Avoid adding an ord atom in math. Use \expandafter's in conditional. \arrayc: Avoid adding an ord atom in math.</td>
</tr>
</tbody>
</table>
v2.5a
\newcolumntype{W}{\dollarbegin and \dolarend so that cell is typeset in mathmode inside \texttt{array}(gh/297)} . . . . . . . . . . . . . . . . 35
\newcolumntype{w}{\dollarbegin and \dolarend so that cell is typeset in mathmode inside \texttt{array}(gh/297)} . . 35

v2.5b
\@yargarraycr: Don’t define \@yargarraycr unnecessarily . 25
\@endpbox: Explicitly run \texttt{par} at the end of pboxes . . . . . . . . . . . 22
\endtabular*: Cancel any outside \texttt{mathsurround}(gh/614) . . . . . 27
\ar@align@ncell: Test against \texttt{strutbox} height (gh/766) . . 19

References

