A new implementation of the array– and tabular–environments

Frank Mittelbach
Johannes Gutenberg Universität
D–6500 Mainz

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 style–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.

Introduction

First we will define the current version of this file:

```
\typeout{Style-Option: 'array' v1.9g \space\space <24.6.88> (F.M.)}
\typeout{English documentation dated \space\space <24.6.88> (F.M.)}
```

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. At the moment we are experimenting with a version where all commands are automatically robust.

The reader should be familiar with the general structure of the environments mentioned above. Further information can be found in LAMPORT [3]. 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.

`p{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}`. In the original definition `p{..}` was a topaligned parbox.
`t{width}`	Equivalent to `\parbox[t]{width}`, the former `p`-option.
`b{width}`	Coincides with `\parbox[b]{width}`.
`>{decl.}`	Can be used before an `l`, `r`, `c`, `p`, `t` 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{..}`, `t{..}` 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 new preamble options.

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 `\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 `\bf`) in a flushed left column, this can be done with `>{\bf}l`. You do not have to begin every entry of the column with `\bf` any more.
- In columns which have been generated with `p`, `t` or `b`, the default value is `\parindent=0pt`. This can be changed with `>{\parindent=1cm}p`.
- 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}
  ```
- 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.

These examples should be sufficient to demonstrate the use of the new preamble options.

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 construction of the preamble

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

token	\@chclass	\@chnum	token	\@chclass	\@chnum
c	0	0	Start	4	—
l	0	1	@-arg	5	—
r	0	2	!	6	—
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

\@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`. This is why the following three lines of code are commented out. Later, throughout the text, I will not mention again explicitly whenever I use a % sign that these parts are already defined in `latex.tex`.

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

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#1{\@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, t 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 #1 is either a c, l or an r.

```
  \@chnum
  \if #1c\z@ \else
   \if #1l\@ne \else
    \if #1r\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 #1|\@ne \else
   \if #1!6 \else
    \if #1@7 \else
     \if #1<8 \else
      \if #1>9 \else
```

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

```
 10
 \@chnum
 \if #1p\thr@@ \else
  \if #1t4 \else
   \if #1b5 \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}
```

Multiple columns (∗–form)

\@xexpast 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* that are to be produced (#2) 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\@thetoksz\relax \let\@thetoks\relax
```

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

```
\def\@tempa{\@thetoksz}%
\ifnum\@tempcnta >0 \@whilenum\@tempcnta >0\do
  {\edef\@tempa{\@tempa\@thetoks}\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\@thetoksz{\the\toks@}\def\@thetoks{\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\@@{}
```

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 problen with token registers. We need one register for every `!` and `@`, while we need two for every `c`, `l`, `r`, `t`, `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 301). The advantage of our approach is the fact that we avoid some of the problems that arise with the other method[1].

[1]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 token register 5. Then we call `\@addtopreamble{\@thetoks5}` where `\@thetoks` is defined in a way that it does not expand (for example it could be equivalent to `\relax`). Every following call of `\@addtopreamble` leaves `\@thetoks5` unchanged in `\@preamble`. If the construction of the preamble is completed we change the definition of `\@thetoks` to `\the\toks` and expand `\@preamble` for the last time. During this process all parts of the form `\@thetoks`⟨*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 `\@thetoks \the\count@` into the preamble. `\@thetoks` 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 304). The macro `\save@decl` saves it into the next free token register, i.e. in `\toks\count@`.

```
\def\save@decl{\toks \count@ = \expandafter
      {\expandafter \relax \@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.

What should happen if we want to add another column to the preamble, i.e. if we have found a `c`, `l`, `r`, `t`, `p` or `b` in the user preamble ? In this case we have the problem that the token register from `>{..}` and `<{..}` has 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` 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.

```
    \@thetoks \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@.

```
      \@thetoks \the\count@}
```

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 that appear 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}
```

The macro \@mkpream

\@mkpream 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.

We deal with the commands \@startpbox and \@endpbox in a similar way, although the reason is different here: these macros expand to 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\@thetoks\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-looking 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\@nextchar
```

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

```
\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 save the value of `\@chclass` in 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
```

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.

```
\or \@acol
```

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`, `t` or `b`. We report an error and ignore the declaration given by `{..}`.

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

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

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

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`, `t` 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 `\@thetoks`. The actual insertion, though, is performed later.

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

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`, `t` 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
```

If the last class was class 1 — meaning 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`.

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

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

```
       \or \@acolampacol
```

Class 3 requires no actions because everything necessary has 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.

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

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

```
       \or \@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 `t`, `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. `t`, `b` or `p`). Since `\@lastchclass` has the value 10 at this point nothing happens when `\@classx` is called. So the macro `\@classz` may start like this:

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

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

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

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

```
       \prepnext@tok
```

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

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

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

```
           \hfil
```

The command `\d@llar` follows expanding into nothing (in the tabular–environment) or into `$`. By providing an appropriate setting of `\d@llar` 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@llar
```

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

```
        \insert@column
```

We end this case with another `\d@llar \hfil`.

```
        \d@llar \hfil
```

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.

```
        \or \d@llar \insert@column \d@llar \hfil
        \or \hfil \d@llar \insert@column \d@llar
```

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

```
        \or $\vcenter
```

The part of the templates which is the same in all three cases (`p`, `t` 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 $%
```

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

```
        \or \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. If so, 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 `\@classx` because afterwards always a new column is started by `c`, `l`, `r`, `p`, `t` 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`. This assures that `<` is treated like `!`.

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

`\@arrayrule` There are only two incompatibilities with the original definition: the p–option mentioned earlier and the definition of `\@arrayrule`. In the original a line without width[2] is created by multiple `\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}
```

[2]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 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
```

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.

```
    \or \@classvii
```

If `\@lastchclass` is 4 or 5 nothing has to be done. Classes 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` (see page 304).

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. This argument should be in math mode if it is used in an array–environment. Therefore we surround it with `\d@llar`'s.

```
    \@addtopreamble{\d@llar\@thetoks\the\count@\d@llar}%
```

Finally we must prepare the next token register.

```
    \prepnext@tok}
```

`\@classi` In the case of class 0 we generated 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 (in case of !{...}) the current token — stored in \@nextchar — has to be inserted into the preamble. This corresponds to calling \@classv.

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

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

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

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

```
  \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 within 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 incorrectly. 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 affects only the depth of the last line.

```
     \vrule \@height \ht\@arstrutbox \@width \z@}
```

\@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{\vrule \@width \z@ \@depth \dp\@arstrutbox \egroup}
```

Building and calling \halign

\@array Now that 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 preamble wanted, 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 built 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
```

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 304). All these now get their actual meaning.

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

With the above defined struts we fix the distance between rows by setting `\lineskip` and `\baselineskip` to 0pt. Since $'s have to be set 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
```

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

```
    \let\\ \@arraycr  \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 `\@arstrut` is easy since we know exactly in which mode TeX is while working on the `\halign` preamble.

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

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 copied directly from the original definition. This is necessary because the `\futurelet` in `\@ifnextchar` might expand a following & token in a construction like `\\ &`. This would otherwise end the alignment template at a wrong time. For further information see [1, Appendix D].

```
  \def\@arraycr{{\ifnum 0=`}\fi
```

Then we test whether the star form is being used and ignore a possible star (I 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\@arstrutbox+ ⟨*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, to hide the \cr in other macros when TeX is skipping conditional text.

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

\@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}}
```

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, t 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 more compactly by writing:

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

After the \@preamble is created we forget all local definitions and contents 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.

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

The Environment Definitions

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

\@halignto \d@llar — In order to conserve the save stack we assign the replacement texts for \@halignto and \d@llar each time globally.

\array — Our new definition of \array then reads:

```
\def\array{\col@sep\arraycolsep
  \gdef\d@llar{$}\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 analogously

```
\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 \gdef\d@llar{}%
```

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 needed any longer.

```
\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
```

Last-minute definitions

If this file is used as a style 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 kinds 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: }%
   \@latexerr{%
   \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 colum-spec. allowed.\fi}\@ehc}                %5
```

`\@tfor` Testing this implementation an error was found in the definition of the LaTeX macro `\@tfor`. It was not implemented according to its specification. The assignment to `\@fortmp` must not take place via `\xdef`. A `\def` has to be used because `#2` should not be expanded. Since this mistake does not show up when `\@tfor` is used in `latex.tex`, it does not seem to have been noticed.

```
\def\@tfor#1:=#2\do#3{\def\@fortmp{#2}\ifx\@fortmp\@mpty
      \else\@tforloop#2\@nil\@nil\@@#1{#3}\fi}
```

References

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

[2] D. E. KNUTH. TeX: The program (Computers & Typesetting Volume B). Addison-Wesley, Reading, Massachusetts, 1986.

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

[4] L. LAMPORT. `latex.tex`, Version 2.09 of ⟨15. Sept. 87⟩.

Index