Une nouvelle écriture des environnements tabular et array de LaTeX*

Frank Mittelbach David Carlisle†

Traduction française par Jean-Pierre Drucbert‡

Imprimé le 20 octobre 2011

Résumé

Ce package est une nouvelle implémentation des environnements `tabular` et `array` de LaTeX. Les principaux avantages de cette implémentation sont de nouvelles options pour mettre en forme les colonnes et que les commandes « fragiles » de LaTeX n'ont plus besoin d'être protégées par `\protect` à l'intérieur de ces environnements.

La majeure partie du code de ce package date de 1988—il en est de même de sa documentation.

1 Introduction

Cette nouvelle implémentation des environnements `tabular` et `array` fait partie d'un projet plus large, dans lequel nous essayons d'améliorer le code de LaTeX et de le rendre encore plus facile à manipuler.

Le lecteur devrait être familier avec les environnements mentionnés ci-dessus. De plus amples informations sont disponibles dans [3] et [1]. Les options supplémentaires qui peuvent être utilisées dans le préambule ainsi que celles qui ont un sens légèrement différent sont décrites dans la table 1.

`\extrarowheight` Il y a un nouveau paramètre, la longueur `\extrarowheight`, qui, si elle est positive, est ajoutée à la hauteur normale de chaque ligne de la table, alors que la profondeur reste la même. Ceci est important pour les tables comportant des lignes horizontales car normalement ces lignes touchent les lettres majuscules. Par exemple, nous avons utilisé `\setlength{\extrarowheight}{1pt}` dans la table 1.

Nous allons présenter quelques exemples des nouvelles options du préambule.

– Si vous désirez utiliser une fonte spéciale (par exemple `\bfseries`) dans une colonne cadrée à gauche, ceci peut se faire avec `>{\bfseries}l`. Vous n'êtes plus obligés de commencer chaque entrée par `\bfseries`.

*Ce fichier a le numéro de version v2.3m, révisé le 1998/05/13.

†David a accepté grâcieusement l'inclusion de code auparavant dans `newarray.sty` pour l'écriture de `\newcolumntype`

‡Dernière mise à jour le 20/01/2000

Options inchangées	
`l`	Colonne cadrée à gauche.
`c`	Colonne centrée.
`r`	Colonne cadrée à droite.
`p{width}`	Équivalente à `\parbox[t]{width}`.
`@{decl.}`	Supprime l'espace entre les colonnes et le remplace par `decl.`.
Options nouvelles	
`m{width}`	Définit une colonne de largeur `width`. Chaque entrée sera centrée en proportion du reste de la ligne. Ceci est assez semblable à `\parbox{width}`.
`b{width}`	Équivalente à `\parbox[b]{width}`.
`>{decl.}`	Peut être utilisée avant une option `l`, `r`, `c`, `p`, `m` ou `b`. Ceci insère `decl.` directement en tête de l'entrée dans la colonne.
`<{decl.}`	Peut être utilisée après une option `l`, `r`, `c`, `p`, `m` ou `b`. Ceci insère `decl.` juste après l'entrée dans la colonne.
`|`	Insère une ligne verticale. La distance entre deux colonnes sera augmentée de l'épaisseur de la ligne, contrairement à la définition originale de LaTeX.
`!{decl.}`	Peut être utilisée n'importe où et correspond à l'option `|`. La différence est que `decl.` sera insérée à la place d'une ligne verticale, donc cette option ne supprime pas l'espace normalement inséré entre les colonnes, contrairement à `@{...}`.

TAB. 1 – Les options du préambule.

– Dans les colonnes qui ont été engendrées par `p`, `m` ou `b`, la valeur par défaut de `\parindent` est 0pt. Ceci peut être modifié par :
`>{\setlength{\parindent}{1cm}}p`.
– les options `<` et `>` avaient été développées à l'origine pour l'application suivante : `>{$}c<{$}` engendre une colonne en mode mathématique dans un environnement tabular. Si vous utilisez ce type de préambule dans un environnement array, vous obtenez une colonne en mode LR car les dollars supplémentaires annulent ceux déjà présents.
– Vous pouvez aussi penser à des applications plus complexes. Un problème classique peut facilement être résolu avec la séquence `>{\centerdots}c <{\endcenterdots}`. Pour centrer des nombres sur leurs points décimaux, il suffit [1] (juste ?) de définir les macros suivantes :

```
{\catcode`\.\active\gdef.{\egroup\setbox2\hbox\bgroup}}
\def\centerdots{\catcode`\.\active\setbox0\hbox\bgroup}
\def\endcenterdots{\egroup\ifvoid2 \setbox2\hbox{0}\fi
```

[1] C'est Frank Mittelbach qui le dit !

```
\ifdim \wd0>\wd2 \setbox2\hbox to\wd0{\unhbox2\hfill}\else
  \setbox0\hbox to\wd2{\hfill\unhbox0}\fi
\catcode`\.12 \box0.\box2}
```

Attention : ces macros ne marchent pas si une cellule comporte plusieurs points décimaux ou si le tableau est utilisé en argument d'une autre commande. Une approche bien meilleure est d'utiliser le package `dcolumn.sty` écrit par David Carlisle.

– En utilisant `c!{\hspace{1cm}}c` vous élargissez d'un centimètre l'espace entre deux colonnes, tandis que `c@{\hspace{1cm}}c` vous donne exactement un centimètre d'espace entre deux colonnes.

1.1 Définir de nouveaux types de colonnes

`\newcolumntype`

Alors qu'il est pratique de taper

`>{`⟨*quelques déclarations*⟩`}{c}<{`⟨*quelques autres déclarations*⟩`}`

si vous avez une colonne particulière dans une seule table, cela devient peu pratique si vous utilisez souvent des colonnes de cette forme. La nouvelle version vous permet de définir un nouveau spécificateur de colonne, par exemple `x`, qui remplacera plusieurs commandes de spécificateurs de colonnes[2]. Nous pouvons donc définir :

`\newcolumntype{x}{>{`⟨ *déclarations*⟩`}{c}<{`⟨*autres déclarations*⟩`}}`

On peut alors utiliser le spécificateur de colonne `x` dans les arguments de préambule de tous les environnements `array` ou `tabular` dans lesquels vous désirez ce format.

Il est habituel d'avoir besoin de colonnes en mode mathématique et en mode LR dans le même tableau. Si nous définissons :

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

alors nous pouvons utiliser `C` pour avoir un mode LR centré dans un `array`, ou un mode mathématique centré dans un `tabular`.

L'exemple donné plus haut pour les « points décimaux centrés » pourrait être affecté à un spécificateur `d` par la commande suivante :

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

Cette solution centre toujours le point dans la colonne. Ceci n'est pas très joli si la colonne est formée de grands nombres, mais avec peu de décimales. Une autre solution d'une colonne `d` est :

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

où les macros adéquates seraient[3] :

```
\def\coldot{.}% Ou si vous pr\'ef\'erez, \def\coldot{\cdot}
```

[2] Cette commande était appelée `\newcolumn` dans `newarray.sty`. Actuellement, la commande `\newcolumn` est toujours valide (mais affiche un message d'avertissement). Dans les versions ultérieures, elle disparaîtra.

[3] Le package `dcolumn.sty` contient des commandes plus robustes basées sur ces idées.

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

Notez que `\newcolumntype` accepte le même argument optionnel que `\newcommand` pour déclarer le nombre d'arguments pour le spécificateur de colonne. Maintenant vous pouvez spécifier `d{2}` dans votre préambule pour une colonne de nombres ayant au plus deux décimales.

Une utilisation assez différente de `\newcolumntype` tire parti du fait que le texte de remplacement dans cette commande peut si nécessaire faire référence à plus d'une colonne. Supposons maintenant que notre document contienne de nombreux environnements `tabular` utilisant le même préambule, mais que vous vouliez en essayer d'autres. La définition originale dans LaTeX (par Leslie Lamport) vous permettait de faire (bien que ce soit sans doute un usage incorrect du système) :

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

`array.sty` fait bien attention de ne pas développer le préambule, et donc la solution ci-dessus ne marche pas avec le nouveau schéma. Cette fonctionnalité est obtenue avec :

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

Le texte de remplacement dans une commande `\newcolumntype` peut faire référence à chacune des primitives de `array.sty` (voir la table 1 en page 2), ainsi qu'aux nouvelles lettres définies dans d'autres commandes `\newcolumntype`.

`\showcols` Une liste de tous les spécificateurs définis par `\newcolumntype` est affichée à l'écran et dans le fichier de logs lorsque la commande `\showcols` est utilisée.

1.2 Variantes spéciales de `\hline`

La famille des environnements `tabular` permet un positionnement vertical basé sur la ligne de base du texte dans lequel l'environnement apparait. Par défaut, l'environnement est centré verticalement, mais cela peut être modifié pour l'aligner sur la première ou la dernière ligne l'environnement en spécifiant la valeur `t` ou `b` à l'argument optionnel de positionnement. Cependant, cela ne marche pas si le premier ou le dernier élément de l'environnement est une commande `\hline` ; dans ce cas, l'environnement est aligné sur la barre horizontale.

Voici un example :

Tables sans et commande hline
tables
avec commandes hline

```
Tables
\begin{tabular}[t]{l}
  sans \\ commande \\ hline
\end{tabular} et \\ tables
\begin{tabular}[t]{|l|}
 \hline
  avec \\ commandes \\ hline \\
 \hline
\end{tabular}
```

`\firsthline` `\lasthline` L'utilisation de `\firsthline` et `\lasthline` va résoudre ce problème, et les tables seront correctement alignées tant que leur première ou dernière ligne ne contient pas d'objets excessivement larges.

Tables sans et commande ...line
tables
avec commandes ...line

```
Tables
\begin{tabular}[t]{l}
  sans \\ commande \\ ...line
\end{tabular} et \\ tables
\begin{tabular}[t]{|l|}
 \firsthline
  avec \\ commandes \\ ...line \\
 \lasrhline
\end{tabular}
```

`\extratabsurround` L'écriture de ces deux commandes contient une longueur supplémentaire, appelée `\extratabsurround`, pour ajouter une espace au début ou à la fin de ces environnements. C'est très utile quand les tables sont emboîtées.

2 Derniers commentaires

2.1 Manipulation des barres de séparation

Il y a deux approches pour la manipulation des barres horizontales et verticales à l'intérieur des tables :

1. ces barres peuvent être placées dans l'espace disponible sans augmenter la taille de la table, ou
2. elles peuvent être placées entre les colonnes et les lignes, et par là même agrandissent la table.

`array.sty` implémente la seconde approche, alors que la première est utilisée dans le noyau de LaTeX. Les deux ont chacunes leurs avantages, mais il faut faire attention aux implications suivantes :

– Avec le LaTeX standard, ajouter des barres dans une table ne modifiera ni la largeur ni la hauteur de la table (à moins d'utiliser des doubles barres) ; par exemple changer le préambule de `lll` en `l|l|l` ne modifie pas le document autrement qu'en ajoutant les barres. Au contraire, avec `array.sty`, une table

qui rentre juste dans la largeur `\textwidth` peut produire un débordement horizontal.

– Avec le LaTeX standard, modifier l'épaisseur des barres peut détériorer l'apparence des tables si on ne modifie pas en même temps les paramètres `\tabcolsep`, etc, l'espace entre la barre et la colonne pouvant alors devenir trop petit (ou trop grand). En fait, la surimpression de texte est même possible. Au contraire, avec `array.sty`, la modification de cette largeur marche souvent sans problème parce que les espaces de séparation (comme `\tabcolsep`, etc) ne dépendent pas de l'épaisseur des barres.

– Avec le LaTeX standard, les tableaux encadrés présentent des coins bizarres parce que les barres horizontales se terminent au milieu des barres verticales. Quand le paramètre `\arrayrulewidth` est grand, l'apparence de la table est très désagréable. Dans ce cas, une simple table comme

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

donnera quelque chose comme

au lieu de

2.2 Comparaisons avec les anciennes versions de `array.sty`

Il y a plusieurs différences dans la façon dont la version 2.1 traite les données incorrectes, même si le fichier source ne semble pas utiliser les fonctionnalités supplémentaires de cette nouvelle version.

– Un préambule de la forme `{wx*{0}{abc}yz}` était traitée dans les versions antérieures à 2.1 comme `{wx}`. La version 2.1 le traite comme `{wxyz}`

– Un paramètre de positionnement erroné comme `[Q]` était considéré comme `[c]` par `array.sty`, mais il est maintenant traité comme `[t]`.

– Un préambule tel que `{cc*{2}}` avec une erreur dans l'argument de l'étoile $*$ engendrera des erreurs différentes dans la nouvelle version. Dans les deux cas, les messages d'erreur ne sont pas d'une aide très utile pour l'utilisateur ordinaire.

– Des répétitions de constructions avec `<` ou `>` engendraient une erreur dans les versions antérieures, mais sont maintenant autorisées. Par exemple, `>{`⟨*decl1*⟩`}>{`⟨*decl2*⟩`}` est équivalent à `>{`⟨*decl2*⟩⟨*decl1*⟩`}`.

– La commande `\extracolsep` ne fonctionne pas avec les anciennes versions de `array.sty` (lire les commentaires dans `array.bug`). Avec la version 2.1, `\extracolsep` peut de nouveau être utilisée dans des expressions `@{...}` comme avec le LaTeX standard ; elle peut aussi se trouver dans une commande `!{...}`, mais il convient de lire la note ci-dessous.

2.3 Bogues et fonctionnalités

– Les messages d'erreur engendrés en décodant les spécifications des colonnes font référence à l'argument du préambule **après** qu'il a été réécrit par la

commande `\newcolumntype`, et non pas au préambule tel qu'il a été écrit par l'utilisateur. Cela semble inévitable pour tout système utilisant un prétraitement, et est donc considéré comme une **fonctionnalité**.

– Le traitement de la déclaration de plusieurs `<` ou `>` peut sembler étrange de prime abord. Les anciennes versions considéraient `>{`⟨*decl1*⟩`}>{`⟨*decl2*⟩`}` comme équivalant à `>{`⟨*decl1*⟩⟨*decl2*⟩`}`. Cependant, cela empêche l'utilisateur de remplacer les paramètres d'une commande `\newcolumntype` définie en utilisant ces déclarations. Par exemple, supposons que nous utilisons un spécificateur de colonne `C` (défini au-dessus) dans un environnement `array`. Le `C` indique du texte centré dans la colonne, alors que `>{\bfseries}C`, qui est réécrit en `>{\bfseries}>{$}c<{$}` ne peut pas spécifier une colonne de texte en gras, parce que le préambule ressemblera en fait à `\hfil$\bfseries$#$ $\hfil` et donc le texte dans la colonne n'est pas affecté par la commande `\bfseries` ! La version actuelle inverse l'ordre de ces déclarations, et ainsi l'exemple précédent fournit un préambule de la forme `\hfil$ $\bfseries#$ $\hfil`, les dollars s'annulant mutuellement pour donner l'effet escompté.
– L'utilisation de `\extracolsep` est sujette aux deux restrictions suivantes. Il doit y a voir au plus une commande `\extracolsep` par spécificateur `@` ou `!`, et cette commande doit être entrée telle quelle, elle ne doit pas être le résultat du développement d'une commande.
Ainsi, `\newcommand{\ef}{\extracolsep{\fill}}` ...`@{\ef}` ne marche pas avec ce package. Cependant, il est possible d'utiliser à la place une construction comme `\newcolumntype{e}{@{\extracolsep{\fill}}`.
– Comme indiqué dans le LaTeX book, afin de définir la commande `\multicolumn`, chaque colonne à l'exception de la première est composée du texte compris dans la cellule et de ce qui se trouve dans l'espace entre les colonnes situé *après* cette colonne. Cela signifie que dans un tableau avec comme préambule `|l|l|l|l|`, une entrée telle que `\multicolumn{2}{|c|}` est incorrecte si elle n'intervient pas dans la première colonne.
Dans le LaTeX standard, cette erreur n'est pas tellement visible parce que cette version ajoute des espaces négatifs, si bien que chaque `|` n'occupe pas de place horizontalement. Mais dans ce package, les lignes verticales conservent leur épaisseur, on voit alors deux lignes si deux barres ont été spécifiées.

3 The documentation driver file

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

```
1 ⟨*driver⟩
2 \NeedsTeXFormat{LaTeX2e}[1995/12/01]
3 \documentclass{ltxdoc}
4
5 \AtBeginDocument{\DeleteShortVerb{\|}}  % undo the default is not used
```

```
6
7  \usepackage{array}
8 ⟨/driver⟩
9 ⟨*driver⟩
10
11  % Allow large table at bottom
12  \renewcommand{\bottomfraction}{0.7}
13
14 \EnableCrossrefs
15  %\DisableCrossrefs   % Say \DisableCrossrefs if index is ready
16
17 \RecordChanges                  % Gather update information
18
19 \CodelineIndex                  % Index code by line number
20
21  %\OnlyDescription    % comment out for implementation details
22  %\OldMakeindex       % use if your MakeIndex is pre-v2.9
23 \begin{document}
24    \DocInput{f-array.dtx}
25 \end{document}
26 ⟨/driver⟩
```

4 The construction of the preamble

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

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

```
27 ⟨*package⟩
28 %\NeedsTeXFormat{LaTeX2e}[1994/05/13]
29 %\ProvidesPackage{array}[\filedate\space version\fileversion]
```

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

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

The currently used token and the one, used before, are needed to decide on how the construction of the preamble has to be continued. In the example mentioned above the `l` causes the preamble to begin with `\hskip\tabcolsep`. Furthermore `# \hfil` would be appended to define a flush left column. The next token is a `c`. Because it was preceded by an `l` it generates a new column. This is done with `\hskip \tabcolsep & \hskip \tabcolsep`. The column which is to be centered will be appended with `\hfil # \hfil`. The token `|` would then add a space of

`\hskip \tabcolsep` and a vertical line because the last token was a `c`. The following token `|` would only add a space `\hskip \doublerulesep` because it was preceded by the token `|`. We will not discuss our example further but rather take a look at the general case of constructing preambles.

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

token	\@chclass	\@chnum	token	\@chclass	\@chnum
c	0	0	Start	4	—
l	0	1	@-arg	5	—
r	0	2	!	6	—
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`, which is the reason why the following three lines of code are commented out. Later throughout the text I will not mention it again explicitely whenever I use a % sign. These parts are already defined in `latex.tex`.

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

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

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

4.1 The character class of a token

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

`\@chclass` and `\@chnum`.

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

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

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

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

```
   \else \z@
```

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

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

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

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

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

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

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

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

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

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

4.2 Multiple columns (∗–form)

\@xexpast
\the@toks
\the@toksz

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

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

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

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

```
\@tempcnta #2
```

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

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

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

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

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

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

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

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

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

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

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

```
70    \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.

```
71 %  \def\@xexnoop#1\@@{}
```

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

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

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

One could also find a solution which only needs two or three token registers by proceeding similarly as in the macro `\@xexpast` (see page 11). The advantage of our approach is the fact that we avoid some of the problems that arise with the other method[4].

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

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

`\count@` We further need a count register to remember which token register is to be used next. This will be initialized with −1 if we want to begin with the token register 0. We use the Plain TeX scratch register `\count@` because everything takes place

[4] Maybe there are also historical reasons.

locally. All we have to do is insert `\the@toks \the \count@` into the preamble. `\the@toks` will remain unchanged and `\the\count@` expands into the saved number.

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

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

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

```
73    \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 14). The macro `\save@decl` saves it into the next free token register, i.e. in `\toks\count@`.

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

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

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

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

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

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

`\@sharp`

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

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

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

Next follows the `#` sign which specifies the place where the text of the column shall be inserted. To avoid errors during the expansions in `\@addtopreamble` we hide this sign in the command `\@sharp` which is temporarily occupied with `\relax`

during the build-up of the preamble. To remove unwanted spaces before and after the column text, we set an `\ignorespaces` in front and a `\unskip` afterwards.

```
77    \ignorespaces \@sharp \unskip
```

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

```
78    \the@toks \the \count@ \relax}
```

5.1 The separation of columns

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

```
79 %    \newif \@iffirstamp
80 %    \def\@addamp{\if@firstamp \@firstampfalse
81 %                 \else \@addtopreamble &\fi}
```

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

```
82 \newdimen\col@sep
83 \def\@acol{\@addtopreamble{\hskip\col@sep}}
84 %    \def\@acolampacol{\@acol\@addamp\@acol}
```

5.2 The macro `\@mkpream`

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

```
85 \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.

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

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

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

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

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

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

```
\prepnext@tok
```

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

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

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

```
{\@testpach
```

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

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

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

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

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

```
\ifcase\@lastchclass
```

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

```
\@acol \or
```

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

```
\or
```

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

```
\@acol \or
```

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

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

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

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

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

```
\or
```

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

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

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

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

6 The macros \@classz to \@classx

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

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

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

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

```
  \ifcase \@lastchclass
```

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

```
  \@acolampacol \or
```

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

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

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

```
112  \@acolampacol \or
```

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

```
113  \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.

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

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

```
115  \@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`.

```
116  \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:

```
117 \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`.

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

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

```
119    \prepnext@tok
```

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

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

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

```
121        \hfil
```

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

```
122        \d@llarbegin
```

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

```
123         \insert@column
```

We end this case with \d@llarend and \hfil where \d@llarend again is either $ or \endgroup.

```
124         \d@llarend \hfil \or
```

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

```
125         \hskip1sp\d@llarbegin \insert@column \d@llarend \hfil \or
126         \hfil\hskip1sp\d@llarbegin \insert@column \d@llarend \or
```

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

```
127     $\vcenter
```

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

```
128     \@startpbox{\@nextchar}\insert@column \@endpbox $\or
```

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

```
129     \vtop \@startpbox{\@nextchar}\insert@column \@endpbox \or
130     \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.

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

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

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

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

```
134         \@classx}
```

\@classviii If the current token is a < the last character class must be 0. In this case it is not necessary to extend the preamble. Otherwise we output an error message, set

`\@chclass` to 6 and call `\@classvi`. By doing this we achieve that `<` is treated like `!`.

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

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

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

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

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

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

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

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

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

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

```
      \@classvii
```

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

```
      \fi}
```

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

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

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

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

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

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

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

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

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

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

Finally we must prepare the next token register.

```
    \prepnext@tok}
```

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

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

Depending on \@chnum a vertical line

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

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

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

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

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

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

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

Our main problem is to obtain the same distance between succeeding lines of the parbox. We have to remember that the distance between two parboxes should be defined by \@arstrut. That means that it can be greater than the distance in a parbox. Therefore it is not enough to set a \@arstrut at the beginning and at the end of the parbox. This would dimension the distance between first and second line and the distance between the two last lines of the parbox wrongly. To prevent

this we set an invisible rule of height `\@arstrutbox` at the beginning of the parbox. This has no effect on the depth of the first line. At the end of the parbox we set analogously another invisible rule which only affects the depth of the last line. It is necessary to wait inserting this strut until the paragraph actually starts to allow for things like `\parindent` changes via `>{...}`.

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

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

```
\def\@endpbox{\@finalstrut\@arstrutbox \egroup\hfil}
```

7 Building and calling `\halign`

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

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

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

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

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

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

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

the opening brace after \ialign has to be implicit as it will be closed in \endarray or another macro.

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

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

What we have not explained yet is the macro \@halignto that was just used. Depending on its replacement text the \halign becomes a \halign to ⟨*dimen*⟩. Now we close the group again. Thus \@startpbox and \@endpbox as well as all token registers get their former meaning back.

```
\endgroup
```

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

```
\@arrayleft
```

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

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

Now another implicit opening brace appears; then definitions which shall stay local follow. While constructing the \@preamble in \@mkpream the # sign must be hidden in the macro \@sharp which is \let to \relax at that moment (see definition of \@mkpream on page 14). All these now get their actual meaning.

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

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

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

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

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

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

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

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

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

8 The line separator \\

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9 Spanning several columns

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

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

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

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

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

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

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

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

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

```
  \endgroup
```

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

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

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

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

10 The Environment Definitions

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

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

In order to relieve the save stack we assign the replacement texts for `\@halignto` globally. `\d@llar` has to be local since otherwise nested tabular and array environments (via `\multicolumn`) are impossible. When the new font selection scheme is in force we have to we surround all `\halign` entries with braces. See remarks in TUGboat 10#2. Actually we are going to use `\begingroup` and `\endgroup`. However, this is only necessary when we are in text mode. In math the surrounding dollar signs will already serve as the necessary extra grouping level. Therefore we switch the settings of `\d@llarbegin` and `\d@llarend` between groups and dollar signs.

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

`\array`

Our new definition of `\array` then reads:

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

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

```
  \@tabarray}
```

`\@tabarray`

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

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

`\tabular` `\tabular*`

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

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

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

```
\expandafter\def\csname tabular*\endcsname#1{%
       \setlength\dimen@{#1}%
       \xdef\@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 \d@llar....

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

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

```
  \@tabarray}
```

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

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

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

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

11 Last minute definitions

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

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

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

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

12 Defining your own column specifiers[6]

\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 2_ε) the old name is not defined.

```
238 ⟨*ncols⟩
```

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

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

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

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

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

```
241   \@ifundefined{NC@find@\NC@char}%
242     {\@tfor\next:=<>clrmbp@!|\do{\if\noexpand\next\NC@char
243         \PackageWarning{array}%
244                       {Redefining primitive column \NC@char}\fi}%
245      \NC@list\expandafter{\the\NC@list\NC@do#1}}%
246     {\PackageWarning{array}{Column \NC@char\space is already defined}}%
```

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

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

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

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

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

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

[6] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`\NC@show` If the column 'C' is defined as above, then `\show\NC@rewrite@C` would output `\long macro: ->\NC@find >{$}c<{$}`. We want to strip the `long macro: ->`

and the \NC@find. So first we use \meaning and then apply the macro \NC@strip to the tokens so produced and then \typeout the required string.

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

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

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

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

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

12.1 The ∗–form

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

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

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

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

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

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

Store the number.

```
  \count@#1
```

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

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

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

```
  \NC@find}
```

12.2 Modifications to internal macros of `array.sty`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
     \or \@classx
     \or \@classx \fi
   \@lastchclass\@chclass}%
   \ifcase\@lastchclass
   \@acol \or
```

```
    \or
    \@acol \or
    \@preamerr \thr@@ \or
    \@preamerr \tw@ \@addtopreamble\@sharp \or
    \or
    \else  \@preamerr \@ne \fi
    \def\the@toks{\the\toks}}
```

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

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

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

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

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

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

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

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

12.3 Support for the `delarray.sty`

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

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

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

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

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

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

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

12.4 Support for \firsthline and \lasthline

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

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

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

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

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

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

We start by producing a single tabular row without any visible content that will produce the external reference point in case [t] is used.

```
\newcommand{\firsthline}{%
  \multicolumn1c{%
```

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

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

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

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

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

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

\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 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.

```
\newcommand{\lasthline}{\hline\multicolumn1c{%
```

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.

```
    \multicolumn1c{%
       \lower\extratabsurround\copy\@arstrutbox
       }%
}
```

12.5 Getting the spacing around rules right

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

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

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

References

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

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

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