blkarray.sty

D. P. Carlisle

27 February 2015

Warning !

> This style option is in the early stages of development. If you want to use an extended `array` or `tabular` in a document, consider using one of the options in the ARRAY package, available from most TEX-servers.
>
> The commands defined in this style are quite likely to have both their user-interface, and their internal definitions changed in later versions.

1 Introduction

This style option implements an environment, `blockarray`, that may be used in the same way as the `array` or `tabular` environments of standard LATEX, or their extended versions defined in `array.sty`. If used in math-mode, `blockarray` acts like `array`, otherwise it acts like `tabular`.

The main feature of this style is that it uses a new method of defining column types. In the simplest form, this has been given a syntax matching the `\newcolumntype` command of `array.sty`.

```
\BAnewcolumntype{C}{>{\large}c}
```

defines a column of large centred text.

In `array.sty` column specifiers defined via `\newcolumntype` are re-written in a preliminary stage to the primitive types, which are then treated by a completely different mechanism (basically a nested `\if` testing each token against one of the predefined column types, `c`, `l`, `>`, . . .

In `blockarray.sty`, *all* column specifiers have equal standing, most of the specifiers of Lamport's original are defined using `\BAnewcolumntype`, e.g.

```
\BAnewcolumntype{c}   {>{\hfil}<{\hfil}}
```

There are one or two other features built into `blockarray`, these will be introduced in no particular order.

1.1 Explicit Column Separators in the Preamble

As described in the LATEX book, originally specifiers like `|` and `@`-expressions were always considered to be part of the *preceding* column (except for expressions before the first column). This can be inconvenient if that column type is going to be over ridden by a `\multicolumn` specification, consider:

```
\begin{tabular}{c|c|c}
 11 & 22                    & 33 \\
 1  &\multicolumn{1}{l|}{2} &  3 \\
 11 & 22                    & 33
\end{tabular}
```

11	22	33
1	2	3
11	22	33

The rule needs to be specified again in the `\multicolumn` argument as `{l|}`, `blockarray` lets you move the rule into the third column, by specifying `&` in the preamble like so:

```
\begin{blockarray}{c|c&|c}
 11 & 22                     & 33 \\
 1  &\BAmulticolumn{1}{l}{2} &  3 \\
 11 & 22                     & 33
\end{blockarray}
```

11	22	33
1	2	3
11	22	33

I first came across the idea of having such a feature in an array preamble when Rainer Schöpf gave a brief introduction to various enhanced array styles. An implementation by Denys Duchier had a feature like this, however I have not seen that style so I do not know the details.

1.2 Blocks

Sometimes you want whole blocks of the table to have a different format, this is often the case with headings for instance. This can be accomplished using lots of `\multicolumn` commands, but this style lets you specify the format for such a block in the usual syntax for column specifiers:

```
\begin{blockarray}{*{3}{c}}
 11 & 22 & 33 \\
 1  & 2  &  3 \\
\begin{block}{*{3}{>{\bf}l}}
 11 & 22 & 33 \\
 1  & 2  & 3  \\
\end{block}
 1  & 2  &  3
\end{blockarray}
```

11	22	33
1	2	3
11	**22**	**33**
1	**2**	**3**
1	2	3

1.3 Delimiters

People often want to put delimiters around sub-arrays of a larger array, delimiters can now be specified in the preamble argument:

```
\begin{blockarray}{[cc]c\}}
 11 & 22  & 33 \\
 1  & 2   &  3 \\
\begin{block}{(ll)l\}}
 11 & 22 & 33 \\
 1  & 2  & 3  \\
\end{block}
 1  & 2   &  3
\end{blockarray}
```

$$\begin{array}{c} \left[\begin{array}{cc} 11 & 22 \\ 1 & 2 \end{array}\right] \left.\begin{array}{r} 33 \\ 3 \end{array}\right\} \\ \left(\begin{array}{ll} 11 & 22 \\ 1 & 2 \end{array}\right) \left.\begin{array}{l} 33 \\ 3 \end{array}\right\} \\ \left[\begin{array}{cc} 1 & 2 \end{array}\right] \left.\begin{array}{r} 3 \end{array}\right\} \end{array}$$

Note how in the previous example the nested `block` was not spanned by the [ ]. each section of the 'outer' block was separately bracketed. If instead of the `block` environment we use `block*`, then the outer brackets will span the inner block, however it is not possible to specify any delimiters in the argument of `block*`.

```
\begin{blockarray}{[cc]c\}}
 11 & 22  & 33 \\
 1  & 2   &  3 \\
\begin{block*}{lll}
 11 & 22 & 33 \\
 1  & 2  & 3  \\
\end{block*}
 1  & 2   &  3
\end{blockarray}
```

$$\left[\begin{array}{cc} 11 & 22 \\ 1 & 2 \\ 11 & 22 \\ 1 & 2 \\ 1 & 2 \end{array}\right] \left.\begin{array}{c} 33 \\ 3 \\ 33 \\ 3 \\ 3 \end{array}\right\}$$

The delimiters, `( ) [ ] \{ \}` have been pre-defined as column specifiers, however any delimiter, including these ones can be specified using the specifiers `\Left` and `\Right`

`\Left{`⟨*text*⟩`}{`⟨*delimiter*⟩`}`

specifies a delimiter together with a 'label' which will be vertically centred with respect to the block. Note that the delimiter and the label take up no horizontal space, and so extra space must be left with a `!-` or `@`-expression or the text will over-print adjacent columns.

1.4 Automatic Numbering

A column specifier `\BAenum` specifies that the row number is to be printed (in a `!`-expression) at that point in each row, this number may be accessed with `\label` in the usual way. The number is a standard LaTeX counter, `BAenumi`, and so the appearence may be changed by altering the default definition of `\theBAenumi`.

1.5 Footnotes

The `\footnote` command may be used inside `blockarray`. Two styles are supported. If the test `BAtablenotes` is true (by default, or after `\BAtablenotestrue`) then footnotes will appear at the end of the table, with lines set to the width of the table. If `BAtablenotes` is false, footnotes within the table will be treated as standard footnotes, with the text (usually) appearing at the foot of the page.

If table notes are being set, the footnote counter is reset at the start of the table. Also an extended version of `\footnotetext` can be used. As described in the book,

`\footnotetext[2]{xxx}` will produce a text marked with the footnote symbol for '2'. However for tablenotes, the optional argument may also be any non-numeric text, in which case it is set directly at the start of the footnote text. So you can go `\footnotetext[\sc source:]{xxx}` or `\footnotetext[\sc note:]{xxx}` anywhere in the table body, before the first numbered footnote.

If `BAtablenotes` is false the footnote text will not appear at the foot of the page if the whole `blockarray` environment is in an environment which treats footnotes in a special way (eg another `blockarray`). So if you have a complicated table which requires tablenotes, but for TeXnical reasons you wish to enter it in the `.tex` file as nested `blockarray` environments, you may set `\BAtablenotestrue` for the outer environment, and then locally set it to false before each of the nested environments. This will ensure that footnotes from all parts of the table will be collected together at the end.

This table is set with `\BAtablenotestrue`.

ONE		TWO[*]	
l-one	l-two	r-one	r-two
l-three[*]	l-four	r-three[*]	r-four
source: Chicago Manual of Style. note: The above attribution is incorrect. [*] Footnote to l-three.		source: Chicago Manual of Style. note: The above attribution is incorrect. [*] Footnote to r-three	
THREE[†]		FOUR	

source: Chicago Manual of Style.

[*] Note on TWO. This is a reasonably long footnote, to show the line breaking.

[†] Note on THREE.

In this example, the outer table is set with `\BAtablenotestrue`, but each of the inner tables is set with a local setting `\BAtablenotesfalse`.

Also the footnotes have been set in a single paragraph. Tablenotes will be set 'run in' a paragraph, after a `\BAparfootnotes` declaration.

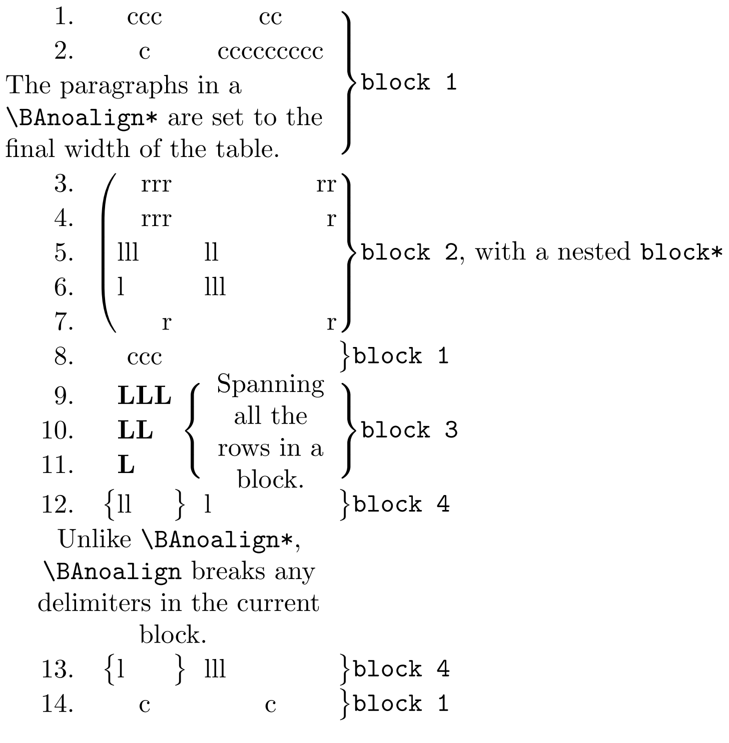
ONE		TWO[*]	
l-one	l-two	r-one	r-two
l-three[†]	l-four	r-three[‡]	r-four
THREE[§]		FOUR	

source: Chicago Manual of Style. note: The above attribution is incorrect. [*]Note on TWO. This is a reasonably long footnote, to show the line breaking. [†]Footnote to l-three. [‡]Footnote to r-three [§]Note on THREE.

1.6 Non Aligned Material

The primitive `\noalign` command may be used with standard LaTeX arrays, but paragraphs inside `\noalign` are broken into lines that are the width of the page, (or at least the current value of `\hsize`) not to the final width of the table. Within a `blockarray` `\BAnoalign` specifies material to be packaged into a parbox the same width as the table. This makes a 'hole' in the current block. `\BAnoalign*` is similar, but any delimiters in the current block span across the non-aligned paragraphs.

```
\begin{blockarray}{\BAenum!{.\quad}cc\Right{\}}{\tt block 1}}
\BAnoalign*{... The paragraphs ...}
\begin{block}{\BAenum!{.\quad}(rr\Right{\}}{{\tt block 2} ...}}
\begin{block*}{\BAenum!{.\quad}(ll)}
\begin{block}{\BAenum!{.\quad}>{\bf}l\{c\Right{\}}{\tt block 3}}
\BAmultirow{50pt}{... Spanning ...}
\begin{block}{\BAenum!{.\quad}\{l\}l\Right{\}}{\tt block 4}}
\BAnoalign{\centering Unlike  ...}
```



```
    1.   ccc         cc        }
    2.    c     ccccccccc      }
 The paragraphs in a           } block 1
 \BAnoalign* are set to the    }
 final width of the table.     }
    3.  ( rrr           rr )   }
    4.  ( rrr            r )   }
    5.  ( lll    ll        )   } block 2, with a nested block*
    6.  ( l      lll       )   }
    7.  (   r            r )   }
    8.   ccc                   } block 1
    9.   LLL { Spanning        }
   10.   LL  {  all the        } block 3
   11.   L   { rows in a       }
               block.
   12.  {ll  }  l              } block 4
   Unlike \BAnoalign*,
  \BAnoalign breaks any
  delimiters in the current
          block.
   13.  {l   }  lll            } block 4
   14.    c          c         } block 1
```

1.7 Spanning Rows and Columns

The previous table had an example of the use of a `\BAmultirow` command. If an entry contains
`\BAmultirow{`⟨*dimen*⟩`}{`⟨*par-mode material*⟩`}`
then the ⟨*par-mode material*⟩ will appear in a box at least ⟨*dimen*⟩ wide, spanning all the rows in the current block. If the other entries in that column of the current block are not empty, they will be over printed by the spanning text.

There is a column specification corresponding to `\BAmultirow`. if
`\BAmultirow{`⟨*dimen*⟩`}`
appears in the preamble, then each entry in that column will be packaged as a paragraph, in a box at least ⟨*dimen*⟩ wide, spanning all the rows in the current block. If this is the last column in the block, you can not use the optional argument to `\\`, and no entry in the column must be empty, it must at least have `{}` in it. (If you need to ask why, you don't want to know!)

Similarly there is a column specification corresponding to `\BAmulticolumn`. if
`\BAmulticolumn{`⟨*number*⟩`}{`⟨*column specification*⟩`}`
appears in the preamble to a `block`, then the rows in the block should have less entries than the outer block, the columns will line up as expected.

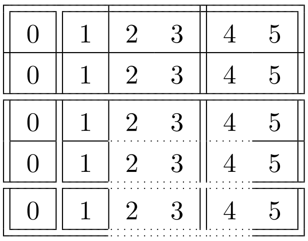
```
\begin{blockarray}{r|lccr|c}
aaa&bbb&ccc&ddd&eee&fff\\
\begin{block}{(r|\BAmulticolumn{4}{>{\bf}l}|c)}
111&The second entry in each &333\\
\end{block}
a&b&c&d&e&f\\
\begin{block}{[r|lccr\{\BAmultirow{1in}]}
111&222&333&444&555&Each entry\\
1&2&3&4&5&in this column is packaged as a paragraph.\\
1&2&3&4&5&\relax\\
\end{block}
a&b&c&d&e&f
\end{blockarray}
```

aaa	bbb	ccc	ddd	eee	fff
a	b	c	d	e	f
111	**The second entry in each row of this block spans 4 columns of the `blockarray`.**				333
1					3
1					3
a	b	c	d	e	f
a	b	c	d	e	f
111	222	333	444	555	Each entry in this column is packaged as a paragraph.
1	2	3	4	5	
1	2	3	4	5	
1	2	3	4	5	
1	2	3	4	5	
a	b	c	d	e	f

1.8 Horzontal Lines

For technical reasons (explained in the code section) the standard `\hline` does not work with `blockarray`. `\BAhline` may be used in just the same way, although currently it is implemented using...
`\BAhhline`. The `\hhline` from `hhline.sty`, would work, but this is a new implementation, more in the spirit of this style.

```
\begin{blockarray}{||c||c&|cc||cc||}
\BAhhline{|t:=:t:=&|==#==:t|}
0&  1  & 2   &  3 &4&5\\
\BAhline
0&  1  & 2   &  3&4&5\\
\BAhline\BAhline
0&  1  & 2   &  3&4&5\\
\BAhhline{||-||-..||.-}
0&  1  & 2   &  3&4&5\\
\BAhhline{=::=""::"=}
0&  1  & 2   &  3&4&5\\
\BAhhline{|b:=:b:=""::"=:b|}
\end{blockarray}
```



0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	5

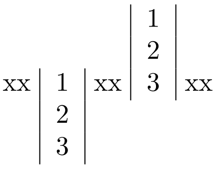
Both `\hline` and `\hhline` increase the (minimum) height of the following row by `\BAextraheightafterhline`, which defaults to 0pt. `array.sty` introduced a parameter, known in this style as `\BAextrarowheight`, which is a length added to the default height of *all* the rows. One of the stated reasons for introducing this was to stop horizontal lines touching large entries like accented capitals, however increasing all the row heights has an effect rather similar to setting `\arraystretch`. This style allows the the extra height just to be added after the horizontal rule.

1.9 Further Thoughts

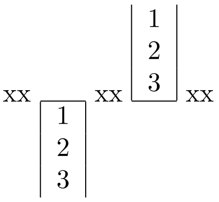
- The main point of any environment based on `\halign` is to make entries line up. Using this style as currently implemented, it is easy to spoil this alignment by putting different `@` expressions or rules in the same column in different blocks. In practice, if you want different `@` expressions, you need to do boxing tricks to make sure that they all have the same width. This could be done automatically by the `\halign`, if the `@`-expressions and rules were put in a separate column. (This fact could be hidden from the user, by a method similar to the multicolumn column specification).

- The `[tcb]` optional argument does not really work at present, I have not done a proper implementation, as I do not know what to do about horizontal rules.

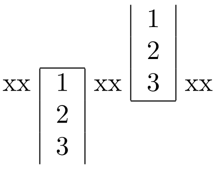
  Standard LaTeX lines `[t]` and `[b]` up like this:

  However if there are horizontal lines, it looks like:

  I *think* I want it to look like this:

  This would be reasonably easy to achieve in a ‘full’ `blockarray`, as each row is taken off and inspected, however I would like an array that only uses the features of the original implementation to be processed by the ‘quick’ system. Any ideas?

- Many user-level commands and parameters defined in this style are named `\BA`... This is to avoid clashes with the standard environments, especially if these are nested inside `blockarray`. If `array` and `tabular` were re-defined in terms of `blockarray`, many commands could be renamed, for example, `\BAextrarowheight`, `\BAmulticolumn`, `\BAhline`.

- This style uses a lot of macros, and every use of the `blockarray` uses a lot more. Does it work at all on a PC?

2 The Macros

```
\ProvidesPackage{blkarray}[2015/02/27 v0.07 Block array (dpc)]\relax
```

2.1 Some General Control Macros

The macros in this section do not have the `BA` prefix, but rather the `GC` prefix, other style files can repeat these definitions without using up TEX's memory.

LATEX provides `\z@`, `\@ne`, `\tw@`, `\thr@@`, but I needed some more…

```
\chardef\GC@four=4
\chardef\GC@five=5
\chardef\GC@six=6
```

2.1.1 Tests

Tests are like `\ifs` except that instead of the
`\if`… ⟨*true-text*⟩`\else`⟨*false-text*⟩`\fi`
notation, they have
`\test`…`{`⟨*true-text*⟩`}{`⟨*false-text*⟩`}`
They are constructed such that they expand directly to either the ⟨*true-text*⟩ or ⟨*false-text*⟩, without leaving a trailing `\fi`.

```
\def\GC@newtest#1{%
  \@namedef{#1true}%
    {\expandafter\let\csname test#1\endcsname\GC@true}%
  \@namedef{#1false}%
    {\expandafter\let\csname test#1\endcsname\GC@false}%
  \@nameuse{#1false}}
```

```
\def\GC@def@testfromif#1#2\fi{%
  \def#1##1##{#2##1\expandafter\GC@true\else\expandafter\GC@false\fi}}
```

```
\def\GC@true#1#2{#1}
\def\GC@false#1#2{#2}
```

This `\testGC@num` is not very good as it does not delimit the ⟨*number*⟩s correctly.

```
\GC@def@testfromif\testGC@x\ifx\fi
\GC@def@testfromif\testGC@num\ifnum\fi
```

2.1.2 List Macros

If `\X` is `abc` then `\GC@add@to@front\X{xyz}` is `xyzabc`;

```
\long\def\GC@add@to@front#1#2{%
  \def\@tempa##1{\gdef#1{#2##1}}%
  \expandafter\@tempa\expandafter{#1}}
```

and `\GC@add@to@end\X{xyz}` is `abcxyz`.

```
\long\def\GC@add@to@end#1#2{%
  \expandafter\gdef\expandafter#1\expandafter{#1#2}}
```

2.2 Allocations

I have given 'meaningful names' to plain-TeX's scratch registers, I am not sure this was a good idea, but it should be OK as long as I always access by name, and do not use, say, `\count4` as a scratch register. I do not like using numbered registers in the code, and can not afford to allocate registers just to get nice names, they are in too short supply already!

Only allocate another register if `blockarray` is going to lose control at a point where the register value needs to be saved. (eg inside a `\BAnoalign` anything can happen.

`\BAtracing` can be set to any integer, the higher the number, the more gets printed.

```
⟨*tracing⟩
\chardef\BAtracing=0
⟨/tracing⟩

\newcounter{BAenumi}\let\BA@row\c@BAenumi
\countdef\BA@row@shadow=6

\countdef\BA@ftn@shadow=0

\newcount\BA@col
\countdef\BA@col@shadow=2

\newcount\BA@block@cnt
\countdef\BA@block@cnt@shadow=4

\countdef\BA@col@max=8

\newbox\BA@final@box
\chardef\BA@final@box@shadow=8
```

v0.07 use newbox not 0 to avoid amsmath.

```
\newbox\BA@first@box

\chardef\BA@tempbox@a=2

\chardef\BA@tempbox@b=4

\chardef\BA@block@box=6

\newdimen\BA@colsep
\BA@colsep=\tabcolsep

\newtoks\BA@ftn
\toksdef\BA@ftnx@shadow=0
```

2.3 'Local' Variables

Most of `blockarray` happens inside a `\halign` which means that the different parts have to make global assignments if they need to communicate. However many of these assignments are logically local to `blockarray`, or a sub-environment like `block`. This means that I have to manage the saving and restoring of local values 'by hand'.

Three different mechanisms occured to me, I have used all of them in this style, mainly just to try them out!

- ‘shadowing’ If `\X` is to be assigned globally, but it is to be considered local to a block of code that corresponds to a TeX group, then it may be shadowed by a local variable `\Y`

  ```
  \begingroup\Y=\X
  \begingroup
  ⟨arbitrary code making global assignments to X⟩
  \endgroup
  \global\X=\Y\endgroup.
  ```

  The inner group is needed to protect `\Y` from being changed.

  This is effectively the situation in the `blockarray` environment, where the outer group is provided by `\begin`. . . `\end`, and the inner group is provided by an assignment to a box register.

- Generating new command names, according to nesting depth. Instead of directly using `\X`, the variable can always be indirectly accessed by `\csname\nesting X\endcsname`. Here `\nesting` should expand to a different sequence of tokens for each nested scope in which `\X` is used. `\nesting` might be altered by a local assignment, or sometimes need to be globally incremented at the start of the scope, and globally decremented at the end.

- Maintaining a stack of previous values. Corresponding to a macro, `\X`, is a macro`\Xstack` which consists of a list of the values of `\X` in all outer environments. When the local scope ends, this stack is popped, and the top value (which was the value of `\X` before the environment) is globally assigned to `\X`.

The first method has the advantage that the variable is normally accessed within the environment, and the code to restore previous values is trivial. The main memory usage is in the save-stack, TeX’s normal place for saving local vaues.

Shadowing can only be used when the environment corresonds to a TeX group. The `block` environment does not!, `\end{block}` is not in the scope of any local assignments made by `\begin{block}`.

The second method, has the advantage that, once the access functions are defined, it is easy to declare new local variables, however unless you keep track of what has been produced, these variables will continue to take up memory space, even after the environment has ended. `blockarray` at the moment does not do much clearing up, so after a `blockarray` there are typically five macros per column per block (u-part, v-part, left right and ‘mid’ delimiters) left taking up space. Not to mention macros containing the texts of any non-aligned entries.

An extra ‘`.`’ will locally be added to `\BA@nesting` as each `blockarray` is entered, this is used as described above.

```
\def\BA@nesting{}
```

These two macros help in accessing macros that are ‘local’ to the current value of `\BA@nesting`.

```
\def\BA@expafter#1#2{%
  \expandafter#1\csname BA@\BA@nesting#2\endcsname}
```

```
\def\BA@use#1{\csname BA@\BA@nesting#1\endcsname}
```

These are similar, but for macros which depend on the column and block involved, not just the outer `blockarray` environment.

```
\def\BA@col@expafter#1#2{%
  \expandafter#1%
    \csname BA@\BA@nesting[\BA@use{blocktype},\the\BA@col]#2\endcsname}
\def\BA@col@use#1{%
  \csname BA@\BA@nesting[\BA@use{blocktype},\the\BA@col]#1\endcsname}
```

The following macros manage a stack as described in the third method above.

```
\def\BA@push@blocktype{%
\edef\@tempa{{{\BA@use{blocktype}}}}%
  \BA@expafter\GC@add@to@front{BTstack\expandafter}\@tempa}
```

```
\def\BA@pop@blocktype{%
\BA@expafter\BA@pop@{BTstack}}
```

```
\def\BA@pop@#1{\expandafter\BA@pop@@#1\@@}
```

```
\def\BA@pop@@#1#2\@@{%
  \BA@expafter\gdef{blocktype}{#1}%
  \BA@expafter\gdef{BTstack}{#2}}
```

2.4 The Block Environment

```
\def\BA@beginblock#1{%
  \noalign{%
    \BA@push@blocktype
    \global\advance\BA@block@cnt\@ne
    \penalty\the\BA@block@cnt
    \BA@expafter\xdef{blocktype\expandafter}\expandafter
      {\the\BA@block@cnt}%
    \penalty\@ne
    \global\BA@col=1
    \global\BA@expafter\def{blank@row}{\crcr}%
    \BA@clear@entry
    \global\let\BA@l@expr\@empty\global\let\BA@r@expr\@empty
    \BA@colseptrue
    \BA@parse#1\BA@parseend
    \ifnum\BA@col@max=\BA@col\else
      \@latexerr{wrong number of columns in block}\@ehc\fi
    \global\BA@col\z@}}
```

```
\def\BA@endblock{%\crcr
  \noalign{%
  \BA@pop@blocktype
  \penalty\BA@use{blocktype}%
  \penalty\tw@}}
```

```
\@namedef{BA@beginblock*}#1{%
  \noalign{%
    \BA@push@blocktype
    \global\advance\BA@block@cnt\@ne
    \BA@expafter\xdef{blocktype\expandafter}\expandafter
      {\the\BA@block@cnt}%
    \global\BA@col=\@ne
    \global\BA@expafter\def{blank@row}{\crcr}%
    \BA@stringafter\let\Left\BA@left@warn
    \BA@stringafter\let\Right\BA@right@warn
    \BA@clear@entry
    \global\let\BA@l@expr\@empty\global\let\BA@r@expr\@empty
    \BA@colseptrue
    \BA@parse#1\BA@parseend
    \ifnum\BA@col@max=\BA@col\else
       \@latexerr{wrong number of columns in block*}\@ehc\fi
    \global\BA@col\z@}}
```

```
\def\BA@left@warn#1#2{%
  \@warning{Left delimiter, \string#2, ignored}\BA@parse}
\def\BA@right@warn#1#2{%
  \@warning{Right delimiter, \string#1, ignored}\BA@parse}
```

```
\@namedef{BA@endblock*}{%\crcr
  \noalign{%
  \BA@pop@blocktype}}
```

2.5 Multicolumn

First we have the `\multicolumn` command to be used as in original LaTeX.

```
\def\BAmulticolumn#1#2#3{%
  \multispan{#1}%
  \global\advance\BA@col#1\relax
  \edef\BA@nesting{\BA@nesting,}%
  \BA@expafter\def{blocktype}{0}%
  {\BA@defcolumntype{&}##1\BA@parseend{%
     \@latexerr{\string& in multicolumn!}\@ehc\BA@parse\BA@parseend}%
  \global\BA@expafter\def{blank@row}{\crcr}%
  \BA@clear@entry
  \global\let\BA@l@expr\@empty\global\let\BA@r@expr\@empty
  \BA@colseptrue
  \BA@parse#2\BA@parseend}%
  \BA@strut\BA@col@use{u}\ignorespaces#3\BA@vpart}
```

Now something more interesting, a `\BAmulticolumn` column specification!

```
\def\BA@make@mc#1{%
  \count@#1\relax
  \BA@make@mcX
  \edef\BA@mc@hash{%
    \noexpand\BA@parse
     >{\BA@mc@spans}\noexpand\BA@MC@restore@hash
     \BA@mc@amps\noexpand\BA@MC@switch@amp}}
```

```
\def\BA@make@mcX{%
  \ifnum\count@=\@ne
```

```
    \def\BA@mc@spans{\null}%
    \let\BA@mc@amps\@empty
  \else
   \advance\count@\m@ne
    \BA@make@mcX
    \GC@add@to@end\BA@mc@spans{\span}%
    \GC@add@to@end\BA@mc@amps{&@{}}%
  \fi}
```

2.6 \BAmultirow

First as a command.

```
\long\def\BAmultirow#1{\kern#1\relax
  \global\BA@quickfalse
  \BA@col@expafter\gdef{mid}}
```

Then as a column specification. (The actual `\BAnewcolumn` comes later)

```
\def\BA@mrow@bslash#1{%
  \kern#1\relax
  \global\BA@quickfalse
  \iffalse{\else\let\\\cr\fi\iffalse}\fi
  \BA@mrow}
```

```
\long\def\BA@mrow#1\BA@vpart{%
  \BA@col@expafter\GC@add@to@end{mid}{\endgraf#1}
  \BA@vpart}
```

2.7 \BAnoalign

```
\def\BAnoalign{%\crcr
  \noalign{\ifnum0=`}\fi
     \global\BA@quickfalse
     \penalty\the\BA@row
  \@ifstar
      {\penalty\GC@four\BA@noalign}%
      {\penalty\BA@use{blocktype}\penalty\thr@@\BA@noalign}}
```

```
\long\def\BA@noalign#1{%
  \long\BA@expafter\gdef{noalign\the\BA@row}{#1}%
  \ifnum0=`{\fi}}
```

2.8 \\

The following code is taken directly from `array.sty`, apart from some name changes. It is very similar to the version in `latex.tex`. Making `\\` into a macro causes problems when you want the entry to be taken as a macro argument. One possibility is to `\let \\` be `\span`, and then have the ⟨*u-part*⟩ of a final column parse the optional argument. There is still a problem with `\end{...}`. Note that at the moment this style assumes that `\\` is used at the end of all lines except the last, even before `\begin{block}` or `\end{block}`, this allows spacing to be specified, and also approximates to the truth about what is actually happening. The idea of making it easier to allow entries to be taken as arguments may be a non-starter if `&` is allowed to become a ‘short-ref’ (ie `\active`) character.

```
\def\BA@cr{{\ifnum 0=`}\fi
  \@ifstar \BA@xcr \BA@xcr}
```

```
\def\BA@xcr{\@ifnextchar [%
  \BA@argcr {\ifnum 0=`{\fi}\cr}}
```

```
\def\BA@argcr[#1]{\ifnum0=`{\fi}\ifdim #1>\z@
  \BA@xargcr{#1}\else \BA@yargcr{#1}\fi}
```

```
\def\BA@xargcr#1{\unskip
  \@tempdima #1\advance\@tempdima \dp\@arstrutbox
  \vrule \@depth\@tempdima \@width\z@ \cr}
```

```
\def\BA@yargcr#1{\cr\noalign{%
     \vskip #1}}
```

```
\newdimen\BAextrarowheight
\newdimen\BAextraheightafterhline
\newdimen\BAarrayrulewidth
   \BAarrayrulewidth\arrayrulewidth
```

```
\newdimen\BAdoublerulesep
   \BAdoublerulesep\doublerulesep
```

The B form of the strut is an extra high strut to use after a horizontal rule.

```
\def\BA@strut{\unhcopy\@arstrutbox}
\let\BA@strutA\BA@strut
\def\BA@strutB{\dimen@\ht\@arstrutbox
  \advance\dimen@\BAextraheightafterhline
  \vrule \@height\dimen@ \@depth \dp\@arstrutbox \@width\z@
  \global\let\BA@strut\BA@strutA}
```

2.9 Begin and End

`\begin{block}` is supposed to expand to `\noalign{...`, but the code for `\begin` would place non-expandable tokens before the `\noalign`. Within the `blockarray` environment, redefine `\begin` so that if its argument corresponds to a command `\BA@begin`⟨*argument*⟩, then directly directly expand that command, otherwise do a normal `\begin`. A matching change is made to `\end`.

```
\let\BA@@begin\begin
\let\BA@@end\end
```

```
\def\BA@begin#1{%
  \expandafter\testGC@x\csname BA@begin#1\endcsname\relax
    {\BA@@begin{#1}}%
    {\csname BA@begin#1\endcsname}}
```

```
\def\BA@end#1{%
  \expandafter\testGC@x\csname BA@end#1\endcsname\relax
    {\BA@@end{#1}}%
    {\csname BA@end#1\endcsname}}
```

2.10 The `blockarray` Environment

```
\def\blockarray{\relax
  \@ifnextchar[{\BA@blockarray}{\BA@blockarray[c]}}
```

```
\def\BA@blockarray[#1]#2{%
  \expandafter\let\expandafter\BA@finalposition
    \csname BA@position@#1\endcsname
  \let\begin\BA@begin
  \let\end\BA@end
  \ifmmode
    \def\BA@bdollar{$}\let\BA@edollar\BA@bdollar
  \else
    \setbox\z@\hbox{$$}%set up math for NFSS (2014/10/16)
    \def\BA@bdollar{\bgroup}\def\BA@edollar{\egroup}%
    \let\BA@bdollar\bgroup\let\BA@edollar\egroup
  \fi
  \let\\\BA@cr
```

Currently I use `\everycr` this means that every macro that uses `\halign` that might be used inside a `blockarray` must locally clear `\everycr`. The version of `array` in `array.sty` does this, but not the one in `latex.tex`.

```
  \everycr{\noalign{%
     \global\advance\BA@row\@ne
     \global\BA@col\z@}}%
```

The `\extrarowheight` code from `array.sty`.

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

As explained above various registers which are 'local' to `blockarray` are always accessed globally, and so must be shadowed by local copies, so that the values can be restored at the end.

```
  \BA@col@shadow=\BA@col
    \global\BA@col=\@ne
  \BA@block@cnt@shadow=\BA@block@cnt
    \global\BA@block@cnt\@M
  \BA@row@shadow\BA@row
    \global\BA@row\z@
  \setbox\BA@final@box@shadow=\box\BA@final@box
    \global\setbox\BA@final@box=\box\voidb@x
  \let\BA@delrow@shadow=\BA@delrow
  \let\testBA@quick@shadow\testBA@quick
    \global\BA@quicktrue
```

If we are using tablenotes, shadow the footnote counter (or possibly mpfootnote), and set up the print style for the table notes.

```
  \testBAtablenotes
    {\edef\BA@mpftn{\csname c@\@mpfn\endcsname}%
```

```
     \@namedef{the\@mpfn}{\BA@fnsymbol{\@nameuse{c@\@mpfn}}}%
     \BA@ftn@shadow=\BA@mpftn\global\BA@mpftn\z@
     \BA@ftnx@shadow=\expandafter{\the\BA@ftn}\global\BA@ftn{}%
      }{}%
```

Locally increase `\BA@nesting` so that macros accessed by ‘the second method’ will be local to this environment.

```
  \edef\BA@nesting{\BA@nesting.}%
```

Now start up the code for this block

```
  \BA@expafter\xdef{blocktype}{10000}%
  \BA@expafter\xdef{BTstack}{\relax}%
  \global\BA@expafter\def{blank@row}{\crcr}%
  \setbox\BA@first@box=\vbox{\ifnum0=`}\fi
    \let\@footnotetext\BA@ftntext\let\@xfootnotenext\BA@xftntext
    \lineskip\z@\baselineskip\z@
    \BA@clear@entry
    \global\let\BA@l@expr\BA@bdollar\global\let\BA@r@expr\BA@edollar
    \global\let\BA@l@expr\@empty\global\let\BA@r@expr\@empty
    \BA@colseptrue
    \BA@parse#2\BA@parseend
    \BA@col@max=\BA@col
```

There had to be a `\halign` somewhere, and here it is!

Currently I am using a ‘repeating preamble’ because it was easier, but I think that I should modify the columntypes for `blockarray` (not `block`) so that they construct a preamble with the right number of columns. this would give better error checking, and would give the possibility of modifying the tabskip glue.

```
    \tabskip\z@
    \halign\bgroup\BA@strut%\global\BA@col=\z@
       \BA@upart##\BA@vpart&&\BA@upart##\BA@vpart\cr
       \noalign{\penalty\GC@five}}
```

```
\def\BA@upart{\global\advance\BA@col\@ne\BA@col@use{u}\ignorespaces}
```

```
\def\BA@vpart{\unskip\BA@col@use{v}}
```

```
\def\BA@clear@entry{%
  \global\BA@col@expafter\let{u}\@empty
  \global\BA@col@expafter\let{v}\@empty
  \global\BA@col@expafter\let{left}\relax
  \global\BA@col@expafter\let{mid}\@empty
  \global\BA@col@expafter\let{right}\relax
  \BA@uparttrue}
```

```
\let\BA@fnsymbol=\@fnsymbol
```

The code to place the footnote texts at the foot of the table, each footnote starting on a new line. This will only be activated if `\BAtablenotestrue`.

```
\def\BA@expft[#1]#2{%
  \noindent\strut\ifodd0#11{%
  \edef\@thefnmark
     {\BA@fnsymbol{#1}}\@makefnmark}\else{#1}\fi\ #2\unskip\strut\par}
```

After a `\BAparfootnotes` declaration, the table notes will be set in a single paragraph, with a good chance of line breaks occuring at the start of a footnote.

```
\def\BAparfootnotes{%
   \def\BA@expft[##1]##2{%
    \noindent\strut\ifodd0##11{%
      \edef\@thefnmark{\BA@fnsymbol{##1}}\@makefnmark}\else{##1~}\fi
    ##2\unskip\strut\nobreak
   \hskip \z@ plus 3em \penalty\z@\hskip 2em plus -2.5em minus .5em}}
\def\endblockarray{%
```

At this point, if no delimiters, `\BAnoalign`, or `\BAmultirow` have been used, just finish here, this makes `blockarray` was just about as efficient as `array` If no fancy tricks have been used.

```
   \testBA@quick\BA@quick@end\BA@work@back@up
```

Now we restore the values that have been 'shadowed' by versions that are local to this environment.

```
   \global\BA@block@cnt=\BA@block@cnt@shadow
   \global\BA@col=\BA@col@shadow
   \global\BA@row=\BA@row@shadow
   \global\setbox\BA@final@box=\box\BA@final@box@shadow
   \global\let\BA@delrow=\BA@delrow@shadow
   \global\let\BA@delrow=\BA@delrow@shadow
   \global\let\testBA@quick\testBA@quick@shadow
```

If tablenotes are being used, reset the shadowed list of footnotes. Otherwise execute the list now, to pass the footnotes on to the outer environment, or the current page.

```
   \testBAtablenotes
     {\global\BA@mpftn=\BA@ftn@shadow
      \global\BA@ftn=\expandafter{\the\BA@ftnx@shadow}}
     {\global\BA@ftn\expandafter{\expandafter}\the\BA@ftn}}
```

Here is the 'quick ending': just position the box as specified by `[tcb]`, possibly adding footnotes.

```
\def\BA@quick@end{%
   \crcr
   \egroup% end of halign
   \ifnum0=`{\fi}% end of \BA@first@box
⟨*tracing⟩
      \ifnum\BAtracing>\z@\typeout{Quick blockarray ends \on@line}\fi
⟨/tracing⟩
    \leavevmode\BA@finalposition\BA@first@box}
```

If any delimiters or noaligns have been used, we must take apart the table built in `\BA@first@box`, and reassemble it. This is done by removing the rows one by one, starting with the last row, using `\lastbox`.

```
\def\BA@work@back@up{%
   \BA@use{blank@row}%
   \egroup% end of halign
   \setbox\@tempboxa=\lastbox
   \unskip
```

```
  \BA@getwidths
⟨*tracing⟩
     \ifnum\BAtracing>\z@\typeout{Full blockarray ends \on@line}\fi
⟨/tracing⟩
⟨*tracing⟩
   \ifnum\BAtracing>\thr@@
     \egroup\showbox\BA@first@box
     \setbox\BA@first@box=\vbox\bgroup\unvbox\BA@first@box
   \fi
⟨/tracing⟩
  \setbox\BA@block@box=\box\voidb@x
  \BA@check@pen
  \ifnum0=`{\fi}% end of \BA@first@box
⟨*check⟩
  \dimen@=\ht\BA@first@box\advance\dimen@\dp\BA@first@box
  \ifdim\dimen@>\z@\showthe\dimen@\fi
⟨/check⟩
     \leavevmode\BA@finalposition\BA@final@box}
```

These macros position the final box, they are just first attempts. In particular `[t]` does not work properly because the guard penalty used to terminate the main loop means that `\BA@first@box` always has zero height. Also if the box has been taken apart, `[t]` and `[b]` will cause the position to be based on the *centre* of the corresponding block.

If tablenotes are being used, package the table in a box with the notes.

```
\newdimen\BAfootskip
\BAfootskip=1em
```

```
\def\BA@position@c#1{\hbox{%
     \testBAtablenotes
      {\let\footnotetext\BA@expft
       \hsize\wd#1\@parboxrestore\footnotesize}%
      {}%
     $\vcenter{%
      \unvbox#1%
      \testBAtablenotes
        {\vskip\BAfootskip
         \the\BA@ftn}{}%
      }$}}
```

```
\def\BA@position@t#1{\vtop{%
     \testBAtablenotes
      {\let\footnotetext\BA@expft
       \hsize\wd#1\@parboxrestore\footnotesize}%
      {}%
      \unvbox#1%
      \testBAtablenotes
        {\vskip\BAfootskip
         \the\BA@ftn}{}}}
```

```
\def\BA@position@b#1{%
    \testBAtablenotes
```

```
      {\vbox{%
        \let\footnotetext\BA@expft
        \hsize\wd#1\@parboxrestore\footnotesize
        \unvbox#1%
        \vskip\BAfootskip
        \the\BA@ftn}}%
      {\box#1}}
```

2.11 Fitting the Parts Together

Get the widths of each column. It also faithfully copies the tabskip glue, even though currently this is always 0pt. The width of the column is put into `\BA@delrow` as the argument to the (unexpanded) call to `\BA@lr`.

```
\def\BA@getwidths{%
  \setbox\@tempboxa=\hbox{\unhbox\@tempboxa
    \xdef\BA@delrow{\hskip\the\lastskip}\unskip
    \let\BA@lr\relax
    \loop
    \setbox\BA@tempbox@a=\lastbox
    \skip\z@=\lastskip\unskip
    \ifhbox\BA@tempbox@a
    \xdef\BA@delrow{%
      \hskip\the\skip\z@\BA@lr{\the\wd\BA@tempbox@a}\BA@delrow}%
    \repeat}}
```

The main mechanism by which `blkarray` leaves information to be used 'on the way back' is to leave groups of penalties in the main box. The last penalty in each group (the first to be seen on the way back) is a code penalty, it has the following meanings:
`\penalty 1` — `\begin{block}`
`\penalty 2` — `\end{block}`
`\penalty 3` — `\BAnoalign`
`\penalty 4` — `\BAnoalign*`
`\penalty 5` — `\begin{blockarray}`
Note that the `block*` environment does not produce any penalties, this environment is just as efficient as `\multicolumn`, and does not require the second phase, coming back via `\lastbox`.

Above the code penalty may be other penalties, depending on the code, typically these have the values of the blocktype for the block, or the row number.

```
\def\BA@check@pen{%
  \count@=\lastpenalty\unpenalty
  \ifcase\count@
```

Grumble Grumble. `\lastpenalty` should be `void` if the previous thing was not a penalty, and there should be an `\ifvoid\lastpenalty` or something equivalent to test for this. If the user manages to get a `\penalty0` into the main box, it will just have to be discarded. Actually that is not disastrous, but if a rule, mark, special, insert or write gets into that box `blockarray` will go into an infinite loop. Every class of TEX object should have a `\last`… so that boxes may be taken apart and reconstructed by special styles like this. TEX of course is frozen, so these missing features will never be added (while the system is called TEX).

```
⟨*tracing⟩
     \ifnum\BAtracing>\tw@\typeout{0-???}\fi
⟨/tracing⟩
     \BA@get@row
  \or
⟨*tracing⟩
     \ifnum\BAtracing>\tw@\typeout{1-block}\fi
⟨/tracing⟩
     \BA@expafter\xdef{blocktype}{\the\lastpenalty}\unpenalty
     \ifnum\lastpenalty=\tw@
⟨*tracing⟩
       \ifnum\BAtracing>\tw@\typeout{discarding 2-endblock}\fi
⟨/tracing⟩
       \unpenalty\unpenalty\fi
     \BA@place
  \or
⟨*tracing⟩
     \ifnum\BAtracing>\tw@\typeout{2-endblock}\fi
⟨/tracing⟩
     \BA@expafter\xdef{blocktype}{\the\lastpenalty}\unpenalty
     \BA@place
  \or
⟨*tracing⟩
     \ifnum\BAtracing>\tw@\typeout{3-BAnoalign}\fi
⟨/tracing⟩
     \BA@expafter\xdef{blocktype}{\the\lastpenalty}\unpenalty
     \BA@place
     \count@=\lastpenalty\unpenalty
     \global\setbox\BA@final@box=\vbox{%
       \hsize=\wd\BA@final@box\@parboxrestore
       \vrule \@height \ht\@arstrutbox \@width \z@
       \BA@use{noalign\the\count@}
       \vrule \@width \z@ \@depth \dp \@arstrutbox
       \endgraf\unvbox\BA@final@box}%
  \or
⟨*tracing⟩
     \ifnum\BAtracing>\tw@\typeout{4-BAnoalign*}\fi
⟨/tracing⟩
     \count@=\lastpenalty\unpenalty
     \setbox\BA@block@box=\vbox{%
       \hsize=\wd\BA@final@box \@parboxrestore
       \vrule \@height \ht\@arstrutbox \@width \z@
       \BA@use{noalign\the\count@}
       \vrule \@width \z@ \@depth \dp \@arstrutbox
       \endgraf\unvbox\BA@block@box}%
  \or
⟨*tracing⟩
```

```
     \ifnum\BAtracing>\tw@\typeout{5-blockarray}\fi
⟨/tracing⟩
     \BA@expafter\xdef{blocktype}{10000}
     \BA@place
     \let\BA@check@pen\relax
  \fi
  \BA@check@pen}
```

Move a row of the table from the `\BA@first@box` into the block that is being constructed in `\BA@block@box`.

```
\def\BA@get@row{%
  \skip@=\lastskip\unskip
  \advance\skip@\lastskip\unskip
  \setbox\@tempboxa=\lastbox
  \setbox\BA@block@box=\vbox{%
     \box\@tempboxa
     \vskip\skip@
     \unvbox\BA@block@box}}
```

Place the block that has been constructed in `\BA@block@box`, together with any delimiters, or spanning entries which have been assembled into `\BA@delrow`, into the final table, which is being constructed in `\BA@final@box`.

```
\def\BA@place{%
  \global\setbox\BA@final@box=\vbox{\hbox{%
    \m@th\nulldelimiterspace=\z@
    \dimen\z@=\ht\BA@block@box
    \advance\dimen\z@ by \dp\BA@block@box
    \divide\dimen\z@\tw@
    \dimen\tw@=\dimen\z@
    \advance\dimen\z@ by\fontdimen22 \textfont\tw@
    \advance\dimen2 by-\fontdimen22 \textfont\tw@
    \global\BA@col=\z@
    \delimitershortfall=10pt
    \delimiterfactor=800
    \BA@delrow
    \kern-\wd\BA@block@box
    \ht\BA@block@box=\dimen\z@ \dp\BA@block@box=\dimen\tw@
    \box\BA@block@box}
  \unvbox\BA@final@box}}
```

Place the delimiters or spanning entries in position for one column of the current block.

```
\def\BA@lr#1{%
 \global\advance\BA@col\@ne\relax
 \BA@col@use{left}%
 \BA@col@expafter\ifx{mid}\@empty
   \kern#1
 \else
   \hbox to #1{%
      \setbox\BA@tempbox@a
```

```
       \hbox{\let\BA@mrow@bslash\@gobble\BA@col@use{u}\BA@edollar}%
     \setbox\BA@tempbox@b
       \hbox{\BA@bdollar\BA@col@use{v}}%
    \kern\wd\BA@tempbox@a
    $\vcenter{%
     \hsize=#1
     \advance\hsize-\wd\BA@tempbox@a
     \advance\hsize-\wd\BA@tempbox@b
     \@parboxrestore\BA@col@use{mid}}$%
    \kern\wd\BA@tempbox@b}%
\fi
\BA@col@use{right}}
```

```
\def\BA@leftdel#1#2#3{%
  \llap{%
    {#1}$\left#2\vrule height \dimen\z@ width\z@ \right.$\kern-#3}}
```

```
\def\BA@rightdel#1#2#3{%
  \rlap{%
    \kern-#3$\left.\vrule height \dimen\z@ width\z@ \right#1${#2}}}%
```

2.12 Parsing the Column Specifications

A token `x` in the column specification, is interpreted by `\BA@parse` as `\BA@<x>`. This command is then expanded, which may take further tokens as arguments. The expansion of `\BA@<x>` is supposed to end with a call to `\BA@parse` which will convert the token following any arguments to a control sequence name. The process is terminated by the token `\BA@parseend` as the coresponding command `\BA@<\BA@parseend>` does some 'finishing off', but does not call `\BA@parse`.

There are two commands to help in defining column types.

`\BA@defcolumntype` This takes its parameter specification in the primitive `\def` syntax, and allows the replacement text to be anything.

`\BAnewcolumntype` This takes its parameter specification in LaTeX's `\newcommand` syntax, and applies `\BA@parse` to the *front* of the replacement text. This is intended for users to define their own column types in terms of primitive column types, rather than in terms of arbitrary TeX expressions.

The preamble argument build up various macros as as follows:

Each entry in the `\halign` preamble is of the form

`\BA@upart#\BA@vpart`

`\BA@upart` increments `\BA@col`, and then expands `\BA@col@use{u}`, similarly `\BA@vpart` expands `\BA@col@use{v}`.

`\BA@col@use{u}` is a macro considered local to the current block and column, it is always accessed via `\BA@col@use` or `\BA@col@expafter`. So the preamble entries must result in `\BA@col@use{u}` and `\BA@col@use{v}` being defined to enter any text specified in `@`-expressions, the inter-column space (in the LaTeX tradition, not `\tabskip`), and any declarations in `>` and `<` expressions. Delimiters are not added to these macros as they correspond to the whole block, they are left in the macros `\BA@col@use{left}` and `\BA@col@use{right}` spanning entries from `\BAmultirow` are considered like delimiters, and left in `\BA@col@use{mid}`.

If the test `\testBA@upart` is true, then the ⟨*u-part*⟩ is being built. This consists of three different sections.
1) The left inter-column text, declared in `!-` and `@`-expressions
2) The left inter-column skip.
3) Any declarations specified in `>` expressions.

Suppose `X` is a column type, which may itself be defined in terms of other column types, then the (equivalent) specifications:
`@{aaa}>{\foo}X` and `>{\foo}@{aaa}X`
must result in `aaa`, surrounded by `\bgroup`...`\egroup` or `$`...`$`, being prepended to the *front* of the u-part, and `\foo` being appended to the end, so that it is the innermost declaration to be applied to the entries in that column.

In order to achieve this, `>` expressions are directly added to the u-part, using `\GC@add@to@front`, `@`- and `!`- expressions (and rules from `|`) are added to a scratch macro, `\BA@l@expr`, using `\GC@add@to@end`.

When the next `#` column specifier is reached, the `\BA@l@expr` is added to the front of the u-part, It is separated from the `>`-expressions by a ⟨*hskip*⟩ unless an `@`-expression has occured, either while building the current u-part, or the previous v-part.

Building the v-part is similar.

This procedure has certain consequences,

- `@{a}@{b}` is equivalent to `@{ab}`, or more correctly `@{{a}{b}}`.
- `>{\a}>{\b}` is equivalent to `>{\b\a}`.
- If any `@`-expression occurs between two columns, all `!`-expressions between those columns will be treated identically to `@`-expressions. This differs from `array.sty` where two `!`-expressions are separated by the same skip as the rules specified by `||`.
- If any rule `|` occurs, then a following rule will be preceded by the doubleruleskip, unless a `@` or `!`-expression comes between them. In particular `|&|` specifies a double rule, which looks the same as `||`, but `\multicolumn` commands can be used to remove one or other of the rules for certain entries.

Create a macro name from a column specifier.

```
\def\BA@stringafter#1#2{\expandafter#1\csname BA@<\string#2>\endcsname}
```

Execute the specifier, or discard an unknown one, and try the next token.

```
\def\BA@parse#1{%
  \BA@stringafter\testGC@x{#1}\relax
    {\@latexerr {Unknown column type, \string#1}\@ehc\BA@parse}%
    {\csname BA@<\string#1>\endcsname}}
```

`\def` for column types.

```
\def\BA@defcolumntype#1{%
  \BA@stringafter\def{#1}}
```

`\newcommand` for column types.[1]

```
\def\BAnewcolumntype{\@ifnextchar[{\BA@nct}{\BA@nct[0]}}
```

```
\def\BA@nct[#1]#2#3{%
  \BA@stringafter\@reargdef{#2}[#1]{\BA@parse#3}}
```

These `\if`s will be true if their associated skips are to be added in the current column.

```
\newif\ifBA@colsep
\newif\ifBA@rulesep
```

This is true if we do not need to come back up the array.

```
\GC@newtest{BA@quick}
```

This will be true if building the ⟨*u-part*⟩, and false if building the ⟨*v-part*⟩.

```
\GC@newtest{BA@upart}
```

2.13 'Internal' Column-Type Definitions

`>` expressions:
If we are building the v-part, add a `&`, and try again, so that `c<{\a}>{\b}c` is equivalent to `c<{\a}&>{\b}c`.
Otherwise add the expression to the front of the u-part, i.e., the list being built in `\BA@col@use{u}`. Note that no grouping is added so that the scope of any declaration includes the column entry.

```
\BA@defcolumntype{>}#1{%
  \testBA@upart
   {\BA@col@expafter\GC@add@to@front{u}{#1}%
    \BA@parse}%
   {\BA@parse &>{#1}}}
```

Left delimiters:
Again add a `&` if required, otherwise just save the delimiter and label as the first two arguments of `\BA@left@del` in the macro `\BA@col@use{left}`.

```
\BA@defcolumntype{\Left}#1#2{%
  \testBA@upart
   {\global\BA@quickfalse
    \BA@col@expafter\gdef{left}{\BA@leftdel{#1}{#2}}\BA@parse}%
   {\BA@parse &\Left{#1}{#2}}}
```

Right delimiters: As for Left.

```
\BA@defcolumntype{\Right}#1#2{%
  \testBA@upart
   {\BA@parse ##\Right{#1}{#2}}
   {\global\BA@quickfalse
    \BA@col@expafter\gdef{right}{\BA@rightdel{#1}{#2}}\BA@parse}}%
```

`&` The end of each column specification is terminated by `&`, either by the user explicitly entering `&`, or one being added by one of the other rewrites.

---

[1] Currently does not check that the type is new.

If we are still in the u-part, finish it off with `#`.
Otherwise add another column to the blank row, advance the column counter by one. Finally reset the variables `\BA@use{u|v|left|mid|right}`

```
\BA@defcolumntype{&}{%
  \testBA@upart
   {\BA@parse ##&}%
   {%
    \BA@expafter\xdef{blank@row}{\BA@use{blank@row}\omit&}%
    \global\advance\BA@col\@ne
    \BA@clear@entry
    \BA@parse}}
```

`#` Add a `&` if required.
Otherwise make the u-part of the current column, and the v-part of the previous one.

```
\BA@defcolumntype{#}{%
  \testBA@upart
   {%
```

Add the intercolumn skips unless a `@` expression has occured since the last `#` entry.

```
    \ifBA@colsep
       \GC@add@to@front\BA@r@expr{\BA@edollar\hskip\BA@colsep}%
       \GC@add@to@end\BA@l@expr{\hskip\BA@colsep\BA@bdollar}%
    \else
       \GC@add@to@front\BA@r@expr{\BA@edollar}%
       \GC@add@to@end\BA@l@expr{\BA@bdollar}%
    \fi
```

Go back to the previous column. Add `\BA@r@expr` to the end of `\BA@col@use{v}`.

```
    \global\advance\BA@col\m@ne
    \BA@col@expafter\GC@add@to@end{v\expandafter}\expandafter
          {\BA@r@expr}%
```

Add the total width of any `@` expressions as the third argument in the right delimiter macro, this will be used to move the delimiters past any inter-column material.

```
    \BA@add@rskip
```

Repeat for the u-part, and left delimiter of the current column.

```
    \global\advance\BA@col\@ne
    \BA@col@expafter\GC@add@to@front{u\expandafter}\expandafter
          {\BA@l@expr}%
     \BA@add@lskip
```

Clear these scratch macros ready for the next column.

```
    \global\let\BA@l@expr\@empty\global\let\BA@r@expr\@empty
```

Reset these tests and ifs, ready for the next inter-column material.

```
    \BA@upartfalse
    \BA@rulesepfalse
    \BA@colseptrue
```

Finally look at the next specifier.

```
    \BA@parse}%
   {\BA@parse &##}}
```

`<` Just like `>`.

```
\BA@defcolumntype{<}#1{%
  \testBA@upart
   {\BA@parse ##<{#1}}%
   {\BA@col@expafter\GC@add@to@front{v}{#1}\BA@parse}}
```

`BA` version of `\vline`.

```
\def\BA@vline{\vrule \@width \BAarrayrulewidth}
```

`|` like `!`, except that a `\hskip\BAdoublerulesep` is added for consecutive pairs.

```
\BA@defcolumntype{|}{%
  \testBA@upart
   {\ifBA@rulesep
      \GC@add@to@end\BA@l@expr{\hskip\BAdoublerulesep\BA@vline}%
    \else
      \GC@add@to@end\BA@l@expr{\BA@vline}%
    \fi}%
   {\ifBA@rulesep
      \GC@add@to@end\BA@r@expr{\hskip\BAdoublerulesep\BA@vline}%
    \else
      \GC@add@to@end\BA@r@expr{\BA@vline}%
    \fi}%
  \BA@ruleseptrue
  \BA@parse}
```

`@` identical to `!`, but set `\BA@colsepfalse`.

```
\BA@defcolumntype{@}#1{%
  \testBA@upart
   {\GC@add@to@end\BA@l@expr{\BA@bdollar#1\BA@edollar}}%
   {\GC@add@to@end\BA@r@expr{\BA@bdollar#1\BA@edollar}}%
  \BA@colsepfalse
  \BA@rulesepfalse
  \BA@parse}
```

`!` Just save the expression, and make `\BA@rulesepfalse` so that the next `|` is not preceded by a skip.

```
\BA@defcolumntype{!}#1{%
  \testBA@upart
   {\GC@add@to@end\BA@l@expr{\BA@bdollar#1\BA@edollar}}%
   {\GC@add@to@end\BA@r@expr{\BA@bdollar#1\BA@edollar}}%
  \BA@rulesepfalse
  \BA@parse}
```

`*`: `*{3}{xyz}` just produces `xyz*{2}{xyz}` which is then re-parsed.

```
\BA@defcolumntype{*}#1#2{%
 \count@=#1\relax
 \ifnum\count@>\z@
    \advance\count@\m@ne
    \edef\@tempa##1{\noexpand\BA@parse##1*{\the\count@}{##1}}%
 \else
    \def\@tempa##1{\BA@parse}%
 \fi
 \@tempa{#2}}
```

`\BA@parseend` this is added to the end of the users preamble, it acts like a cross between `#` and `&`. It terminates the preamble building as it does not call `\BA@parse`.

```
\BA@defcolumntype{\BA@parseend}{%
  \testBA@upart
   {\BA@parse ##\BA@parseend}%
   {%
    \ifBA@colsep
       \GC@add@to@front\BA@r@expr{\BA@edollar\hskip\BA@colsep}%
    \else
       \GC@add@to@front\BA@r@expr{\BA@edollar}%
    \fi
    \BA@expafter\xdef{blank@row}{\BA@use{blank@row}\omit\cr}%
    \BA@add@rskip
    \BA@col@expafter\GC@add@to@end{v\expandafter}\expandafter
           {\BA@r@expr}}}
```

Like `array.sty`

```
\def\BA@startpbox#1{\bgroup
  \hsize #1 \@arrayparboxrestore
   \vrule \@height \ht\@arstrutbox \@width \z@}

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

Save the `&` and `#` macros, so they can be restored after a multicolumn, which redefines them.

```
\BA@stringafter{\let\expandafter\BA@save@amp}{&}
\BA@stringafter{\let\expandafter\BA@save@hash}{#}
```

A column specification of `\BAmulticolumn{3}{c}` is re-written to:
`c\BA@MC@end`, except that the definition of `#` has been changed so that it expands to:
`>{\null\span\span}\BA@MC@restore@hash&@{}&@{}\BA@MC@switch@amp`
The 2 `\span`s in the u-part make the entry span 3 columns, the 2 `&@{}` increment `\BA@col` without adding any intercolumn skips. The `\BA@MC@restore@hash` specifier just restores `#` to its normal meaning. `\BA@MC@switch@amp` then causes a specifier `&` to generate an error, as the argument to multicolumn may only specify one column. Finally when `\BA@MC@end` is reached, `&` is restored.

`\BA@MC@restore@hash` restore the meaning of `#`.

```
\BA@defcolumntype{\BA@MC@restore@hash}{%
  \BA@stringafter\let{##}\BA@save@hash
  \BA@parse}
```

Switch the meaning of `&` so it generates an error, and skips all specifiers up to `\BA@MC@end`

```
\BA@defcolumntype{\BA@MC@switch@amp}{%
  \BA@stringafter\let{&}\BA@extra@amp
  \BA@parse}
```

Restore `&`.

```
\BA@defcolumntype{\BA@MC@end}{%
  \BA@stringafter\let{&}\BA@save@amp
  \BA@parse}
```

The special definition of `&` while parsing the multicolumn argument.

```
\def\BA@mc@extra@amp#1\BA@MC@end{%
    \@latexerr{\string& in multicolumn!}\@ehc\BA@parse\BA@MC@end}%
```

Putting it all together!

```
\BA@defcolumntype{\BAmulticolumn}#1#2{%
  \BA@make@mc{#1}%
  \BA@stringafter\let{##}\BA@mc@hash
  \BA@parse#2\BA@MC@end}
```

As explained above, in order to position the delimiters on the way back we need the widths of the inter-column texts.

```
\def\BA@add@rskip{%
  \BA@col@expafter\ifx{right}\relax\else
    \setbox\BA@tempbox@a\hbox{\BA@bdollar\BA@r@expr}%
    \BA@col@expafter\GC@add@to@end{right\expandafter}\expandafter
       {\expandafter{\the\wd\BA@tempbox@a}}\fi}

\def\BA@add@lskip{%
  \BA@col@expafter\ifx{left}\relax\else
    \setbox\BA@tempbox@a\hbox{\BA@l@expr\BA@edollar}%
    \BA@col@expafter\GC@add@to@end{left\expandafter}\expandafter
      {\expandafter{\the\wd\BA@tempbox@a}}\fi}
```

2.14 User Level Column-Type Definitions

```
\BAnewcolumntype{c}   {>{\hfil}<{\hfil}}
\BAnewcolumntype{l}   {>{}<{\hfil}}
\BAnewcolumntype{r}   {>{\hfil}<{}}

\BAnewcolumntype[1]{p}{>{\vtop\BA@startpbox{#1}}c<{\BA@endpbox}}
\BAnewcolumntype[1]{m}{>{$\vcenter\BA@startpbox{#1}}c<{\BA@endpbox$}}
\BAnewcolumntype[1]{b}{>{\vbox\BA@startpbox{#1}}c<{\BA@endpbox}}

\BAnewcolumntype{(}  {\Left{}{(}}
\BAnewcolumntype{)}  {\Right{)}{}}
\BAnewcolumntype{\{} {\Left{}{\{}}
\BAnewcolumntype{\}} {\Right{\}}{}}
\BAnewcolumntype{[}  {\Left{}{[}}
\BAnewcolumntype{]}  {\Right{]}{}}

\BAnewcolumntype{\BAenum}  {%
  !{%
    {\def\protect{\noexpand\protect\noexpand}%
```

```
     \xdef\@currentlabel{\p@BAenumi\theBAenumi}}
    \hbox to 2em{%
    \hfil\theBAenumi}}}
```

```
\BAnewcolumntype[1]{\BAmultirow}{>{\BA@mrow@bslash{#1}}##}
```

2.15 Footnotes

This test is true if footnote texts are to be displayed at the end of the table.

```
\GC@newtest{BAtablenotes}
\BAtablenotestrue
```

Inside the alignment just save up the footnote text in a token register.

```
\long\def\BA@ftntext#1{%
  \edef\@tempa{\the\BA@ftn\noexpand\footnotetext
                   [\the\csname c@\@mpfn\endcsname]}%
  \global\BA@ftn\expandafter{\@tempa{#1}}}%
```

```
\long\def\BA@xftntext[#1]#2{%
  \global\BA@ftn\expandafter{\the\BA@ftn\footnotetext[#1]{#2}}}
```

2.16 Hline and Hhline

The standard `\hline` command would work fine ‘on the way down’ but on the way back it throws me into an infinite loop as there is no `\lastrule` to move the rule into the final box. I could just make `\hline` leave a code penalty, and put in the rule on the way back, but this would mean that every array with an `\hline` needs to be taken apart. I hope to make ‘most’ arrays be possible without comming back up the array via `\lastbox`. I could do something with `\leaders` which are removable, but for now, I just make `\hline` and `\hline\hline` just call `\hhline` with the appropriate argument. The `\hhline` from `hhline.sty` does work, but needs extra options to deal with `&` etc, but here is a re-implementation, more in the spirit of this style.

```
\def\BAhline{%
  \noalign{\ifnum0=`}\fi
   \futurelet\@tempa\BA@hline}
```

```
\def\BA@hline{%
  \ifx\@tempa\BAhline
    \gdef\BA@hline@@##1{\BAhhline{*{\BA@col@max}{=}}}%
  \else
    \gdef\BA@hline@@{\BAhhline{*{\BA@col@max}{-}}}%
  \fi
  \ifnum0=`{\fi}%
  \BA@hline@@}
```

```
\def\BAhhline#1{%
  \omit
```

First set up the boxes used in `\leaders`.

```
\global\setbox\BA@ddashbox=\BA@HHbox\BAarrayrulewidth\BAarrayrulewidth
\global\setbox\BA@dashbox=\hbox to \GC@six\BAarrayrulewidth{%
```

```
  \hss
  \vrule\@height\BAarrayrulewidth \@width\BAarrayrulewidth \@depth\z@
  \hss}%
  \global\let\BA@strut\BA@strutB
  \global\BA@rulesepfalse
  \global\BA@uparttrue
  \BA@HHparse#1\BA@HHend}
\def\BA@HHexp#1#2{\expandafter#1\csname aa\string#2\endcsname}
\def\BA@HHparse{{\ifnum0=`}\fi\BA@HHparsex}
\def\BA@HHparsex#1{\BA@HHexp\aftergroup{#1}\ifnum0=`{\fi}}
\BA@HHexp\def\BA@HHend{%
  \cr}
\newbox\BA@dashbox
\newbox\BA@ddashbox
\BA@HHexp\def|{%
  \ifBA@rulesep\hskip\BAdoublerulesep\fi
  \global\BA@ruleseptrue
  \vrule\@width\BAarrayrulewidth
  \BA@HHparse}
```

`:` denotes a broken vertical rule, as in `hhline.sty`. If the double dots `::::` currently produced by `"` turn out to be useful, it might be better to use `:` for them, and something else, perhaps `!` for this feature.

```
\BA@HHexp\def:{%
  \ifBA@rulesep\hskip\BAdoublerulesep\fi
  \global\BA@ruleseptrue
  \copy\BA@ddashbox
  \BA@HHparse}
\BA@HHexp\def-{%
  \testBA@upart{}{&\omit\global\BA@uparttrue}%
  \leaders\hrule\@height\BAarrayrulewidth\hfil
  \global\BA@upartfalse
  \global\BA@rulesepfalse
  \BA@HHparse}
\BA@HHexp\def.{%
  \testBA@upart{}{&\omit\global\BA@uparttrue}%
  \copy\BA@dashbox\xleaders\copy\BA@dashbox\hfil\copy\BA@dashbox
  \global\BA@rulesepfalse
  \global\BA@upartfalse
  \BA@HHparse}
\BA@HHexp\def"{%
  \testBA@upart{}{&\omit\global\BA@uparttrue}%
  \setbox\z@\hbox to \GC@six\BAarrayrulewidth
      {\hss\copy\BA@ddashbox\hss}%
  \copy\z@\xleaders\copy\z@\hfil\copy\z@
  \global\BA@rulesepfalse
  \global\BA@upartfalse
  \BA@HHparse}

\BA@HHexp\def={%
  \testBA@upart{}{&\omit\global\BA@uparttrue}%
  \copy\BA@ddashbox\xleaders\copy\BA@ddashbox\hfil\copy\BA@ddashbox
  \global\BA@rulesepfalse
  \global\BA@upartfalse
  \BA@HHparse}

\BA@HHexp\def~{%
  \testBA@upart{}{&\omit\global\BA@uparttrue}%
  \hfill
  \global\BA@rulesepfalse
  \global\BA@upartfalse
  \BA@HHparse}

\BA@HHexp\def#{%
  \ifBA@rulesep\hskip\BAdoublerulesep\fi
  \global\BA@ruleseptrue
  \vrule\@width\BAarrayrulewidth
  \BA@HHbox\BAdoublerulesep\BAdoublerulesep
  \vrule\@width\BAarrayrulewidth
  \BA@HHparse}

\BA@HHexp\def{t}{%
  \rlap{\BA@HHbox\BAdoublerulesep\z@}%
  \BA@HHparse}

\BA@HHexp\def{b}{%
  \rlap{\BA@HHbox\z@\BAdoublerulesep}%
  \BA@HHparse}

\def\BA@HHbox#1#2{\vbox{%
  \hrule \@height \BAarrayrulewidth \@width #1
  \vskip \BAdoublerulesep
  \hrule \@height \BAarrayrulewidth \@width #2}}

\BA@HHexp\def&{&\omit\global\BA@uparttrue\BA@HHparse}

\BA@HHexp\def{*}#1#2{%
 \count@=#1\relax
 \ifnum\count@>\z@
    \advance\count@\m@ne
    \edef\next##1{\noexpand\BA@HHparse##1*{\the\count@}{##1}}%
 \else
    \def\next##1{\BA@HHparse}%
 \fi
 \next{#2}}
```