The fcolumn package[*]

Edgar Olthof
edgar <dot> olthof <at> inter <dot> nl <dot> net

Printed September 27, 2015

Abstract

In financial reports, text and currency amounts are regularly put in one table, e.g., a year balance or a profit-and-loss overview. This package provides the settings for automatically typesetting and checking such columns, including the sum line (preceded by a rule of the correct width), using the specifier `f`.

1 Introduction

The package `fcolumn` provides the macros for an extra tabular specifier that makes creating financial tables easy. The column specifier `f` itself is rather simple. It is the predefined version of a generic column `F`. The generic version expects three arguments: `#1` is the group separator, `#2` is the decimal mark, and `#3` the coding used for grouping digits of the integer part and decimal part.

The f-column in the current version of the package is defined for the continental European standard: `\newcolumntype{f}{F{.}{,}{3,2}}`. This means that a number like 1234567 will be typeset as 12.345,67. People in the Anglo-saxon world would rather code `\newcolumntype{f}{F{,}{.}{3,2}}`, yielding 12,345.67 for the number given above. The default value for `#3` is 3,2, indicating that grouping of the integer part is by three digits and that the decimal part consists of two digits. If however, in your country or company grouping is done with a thinspace every four digits and there are three digits after the decimal mark—that happens to be a `\cdot`—, then simply specify `\newcolumntype{f}{F{\,}{\cdot}{4,3}}` in that case.

By default two digits are used for the decimal part, so if you really want no decimal digits (in that case of course also skipping the decimal mark) you have to explicitly specify `x,0`.

This package requires and loads the `array` package [1]. To show where and how the F-column is used, let's look at some typical financial information as shown in Table 1 and how this is entered in LaTeX (Table 2). All the work was done by the column specifier "f" (for "finance"). In this case it constructs the `\sumline`,

[*]This file has version number v1.1.1, last revised 2015/09/20.

Table 1: Example Table

Balance sheet

properties	31 dec 2014	debts	31 dec 2014
house	200.000,00	equity capital	50.000,00
bank account	−603,23	mortgage	150.000,00
savings	28.000,00		
cash	145,85	profit	27.542,62
	227.542,62		227.542,62

Table 2: Verbatim version of Example Table

```
\begin{table}[htb]
\caption{Example Table}
\label{tab:ex1}
\begin{tabular}{@{}lflf@{}}
\multicolumn4c{\bfseries Balance sheet}\\
\toprule
properties&\multicolumn1r{31 dec 2014}&
debts&\multicolumn1{l@{}}{31 dec 2014}\\
\midrule
house &     20000000 & equity capital&  5000000\\
bank account& -60323 & mortgage &      15000000\\
savings&     2800000 \\
cash&          14585 & profit  &        2754262\\
\sumline
\bottomrule
\end{tabular}
\end{table}
```

typesets the numbers, calculates the totals, determines the widths of the sumrules, and checks whether the two columns are in balance; if not, the user is warned via a `\message`. Of course for nice settings the `booktabs` package [2] was used, but that is not the point here.

This package is heavily inspired by the `dcolumn` package by David Carlisle [3], some constructions are more or less copied from that package.

2 Commands

The user only needs to know six commands or constructions. These six are given here.

`F` In the tabular the column specifier `F` can be given with arguments, or the predefined version `f`, where the three arguments of `F` are `{.}`, `{,}`, and `3,2`. If you want `g` to be your own definition like the curious one given in Section 1, then specify `\newcolumntype{g}{F{\,}{\cdot}{4,3}}` prior to using `g` in a tabular.

Entries in an F-column are from that moment on, treated as (integer) numbers unless explicitly escaped by `\leeg`, see below. The numbers are typeset according to the template the user gives with his/her `F`-column.

`\sumline` The numbers in an `F`-column are typeset as a financial amount, but the real benefit comes with the `\sumline`. It does three things:

- It calculates the total of the column so far and the maximum width encountered so far, including the width of the total;
- It generates a rule with width calculated in the first item;
- It checks the columns that are supposed to balance whether or not they actually do. If so, nothing happens. If not, a `\message` is given that column i and j do not balance, where i and j are the relevant columns. This is only done if the total number of `F`-columns is even, e.g., if there are six `F`-columns, then 1 is checked against 4, 2 against 5, and 3 against 6. If the number of `F`-columns is odd then anything could be possible in that table and nothing is assumed about structure within the table. This behaviour can be overridden, see below.

`\resetsumline` Suppose you want to typeset one tabular with the profit-and-loss of many projects individually. The layout of those tabulars is the same and it were nice if all columns were aligned. This can be done by making it one big tabular with a fresh start for each project. The macro `\resetsumlines` is used for that: it resets all totals and all column widths, see for example Table 3. Note that the rules in the first and

Table 3: Example: multiple projects

		Project 1			
expense	actual	budget	income	actual	budget
food	450,00	500,00	tickets	1.000,00	1.000,00
drinks	350,00	400,00			
music	180,00	100,00			
profit	20,00				
	1.000,00	1.000,00		1.000,00	1.000,00
		Project 2			
expense	actual	budget	income	actual	budget
food	250,00	300,00	tickets	400,00	450,00
drinks	100,00	80,00			
music	80,00	70,00	loss	30,00	
	430,00	450,00		430,00	450,00

third F-columns of project 1 cover 1.000,00 whereas in project 2 those rules are shorter since they only cover 430,00; still the columns are aligned. The verbatim way of setting up Table 3 is given in Table 4.

Table 4: Verbatim version of Table 3

```
\begin{table}[htb]
\caption{Example: multiple projects}
\label{tab:ex3}
\begin{tabular}{@{}lfflff@{}}
\multicolumn6c{\bfseries Project~1}\\
\toprule
expense&\multicolumn1r{actual}&\multicolumn1r{budget}&
income&\multicolumn1r{actual}&\multicolumn1{r@{}}{budget}\\
\midrule
food &         45000 & 50000 & tickets&  100000 &  100000\\
drinks &       35000 & 40000 \\
music &        18000 & 10000 \\
profit &        2000 \\
\sumline
\resetsumline
\multicolumn6c{\bfseries Project~2}\\
\toprule
expense&\multicolumn1r{actual}&\multicolumn1r{budget}&
income&\multicolumn1r{actual}&\multicolumn1{r@{}}{budget}\\
\midrule
food &         25000 & 30000 & tickets&  40000 &  45000\\
drinks &       10000 &  8000 \\
music &         8000 &  7000 & loss &     3000 \\
\sumline
\bottomrule
\end{tabular}
\end{table}
```

`\leeg` If an `F`-column should be empty then simply leave it empty. If however it should not be empty but the entry should be treated as text—even it is a number—this can be done with `\leeg`. It expects an argument and this argument is typeset in the column, affecting the maximum column width so far. The common case is where `p.m.` (*pro memoria*) is entered. An empty `F`-column followed by `\\` does not work: either remove the `&` or specify `\leeg{}`.

`\checkfcolumns` The automatic column balance check can also be done manually. If `F`-columns 1 and 4 should balance and you want them to be checked, then simply say `\checkfcolumns14`. With more than nine `F`-columns you may be forced to say something like `\checkfcolumns{10}{12}`.

`\strictaccounting` In the rare occasion that a negative number occurs in a financial table, the sign of that number can be an explicit minus sign (−) or the number is coloured red, or it is typeset between parentheses, and there may be even other ways. By default (for aesthetic reasons) `fcolumn` typesets it with a minus sign, but strict accounting prescibes that the number should be put between parentheses. The latter can be accomplished by setting `\strictaccountingtrue`.

3 The macros

`column F` `column f` The column specifier `F` is the generic one, and `f` is the default (European) one for easy use. Note that the definition of the column type `f` does not use private macros (no `@`), so overriding its definition is easy for a user.

```
\newcolumntype{F}[3]{>{\b@fi{#1}{#2}{#3}}r<{\e@fi}}
\newcolumntype{f}{F{.}{,}{3,2}}
```

`\FCsc@l` `\FCtc@l` Two ⟨*count*⟩s are defined, that both start at zero: the ⟨*count*⟩ `\FCsc@l`, that keeps track at which F-column the tabular is working on and the ⟨*count*⟩ `\FCtc@l`, that records the number of F-columns that were encountered so far. Later in the package the code can be found for generating a new ⟨*count*⟩ and a new ⟨*dimen*⟩ if the number of requested F-columns is larger than currently available. This is of course the case when an F-column is used for the first time.

```
\newcount\FCsc@l \FCsc@l=0
\newcount\FCtc@l \FCtc@l=0
```

`\geldm@cro` The macro `\geldm@cro` takes a number and by default interprets this as an amount expressed in cents (dollar cents, euro cents, centen, Pfennige, kopecks, groszy) and typesets it as the amount in entire currency units (dollars, euros, guldens, Marke, ruble, złoty) with comma as decimal separator and the dot as thousand separator. As explained, this can be changed. It uses a private boolean `\withs@p` and an accessable, i.e., without `@` boolean: `\strictaccounting`. The latter is used to typeset negative numbers between parentheses. By default it doesn't do this: a minus sign is used.

```
\newif\ifwiths@p
\newif\ifstrictaccounting \strictaccountingfalse
```

Actually `\geldm@cro` is only a wrapper around `\g@ldm@cro`.

```
\def\geldm@cro#1#2{\withs@pfalse
\afterassignment\g@ldm@cro\count@#2\relax{#1}}
```

`\g@ldm@cro` This macro starts by looking at the sign of `#2`: if it is negative, it prints the correct indicator (a parenthesis or a minus sign), assigns the absolute value of `#2` to `\count2` and goes on. Note that `\geldm@cro` and therefore `\g@ldm@cro` are always used within `$`s, so it is really a minus sign that is printed, not a hyphen. All calculations are done with `\count0`, `\count1`, etc. i.e., without fcolumn-specific ⟨*count*⟩s because it is all done locally. Leaving the tabular environment will restore their values.

```
\def\g@ldm@cro#1\relax#2{%
\ifnum#2<0 \ifstrictaccounting(\else-\fi\count2=-#2 \else\count2=#2 \fi
```

Calculate the entire currency units: this is the result of x/a as integer division, with $a = 10^n$ and n the part of `#1` after the separator (if any). Here the first character of `#1` is discarded, so the separator in `#1` is not strict: you could also specify `3.2` instead of `3,2` (or even `3p2`).

```
\count3=\ifx\relax#1\relax2 \else \@gobble#1\relax\fi
\count4=\count3
\loop
 \ifnum\count3>0 \divide\count2 by 10 \advance\count3 by \m@ne
\repeat
```

The value in `\count2` is then output by `\g@ldens` using the separation given.

```
\g@ldens{\the\count@}%
```

If there is a decimal part...

```
\ifnum\count4>0\decim@lmark
```

Next the decimal part is dealt with. Now x mod a is calculated in the usual way: $x - (x/a) * a$ with integer division. The minus sign necessary for this calculation is introduced in the next line by changing the comparison from `<` to `>`.

```
 \ifnum#2>0 \count2=-#2\else\count2=#2 \fi
 \count3=\count4
 \loop
  \ifnum\count3>0 \divide\count2 by 10 \advance\count3 by \m@ne
 \repeat
```

The value of `\count3` is 0 now. Now counting up again, this saved an assignment.

```
 \loop
  \ifnum\count3<\count4 \multiply\count2 by 10 \advance\count3 by \@ne
 \repeat
 \ifnum#2>0 \advance\count2 by #2
 \else \advance\count2 by -#2
 \fi
 \zerop@d{\number\count3}{\number\count2}%
\fi
```

If the negative number is indicated by putting it between parentheses, then the closing parenthesis should stick out of the column, otherwise the alignment of this entry in the column is wrong. This is done by an `\rlap` and therefore does not influence the column width. For the last column this means that this parenthesis may even stick out of the table. I don't like this, therefore I chose to put `\strictaccountingfalse`. Change if you like.

```
\ifnum#2<0 \ifstrictaccounting\rlap{)}\fi\fi}
```

`\g@ldens` Here the whole currency units are dealt with. The macro `\g@ldens` is used recursively, therefore the double braces; this allows to use `\count0` locally. Tail recursion is not possible here (at least I don't know of a way), but that is not very important, as the largest number (which is $2^{31} - 1$) will only cause a threefold recursion. This also implies that the largest amount this package can deal with is 2.147.483.647 (using `x.0`). For most people this is probably more than enough if the currency is euros or dollars. If not, then it is more likely that you are spending your time on the beach than studying this package. And otherwise make clear that you use a currency unit of k$.

```
\def\g@ldens#1{{\count3=\count2 \count0=#1
```

First divide by 10^n, where n is `#1`.

```
\loop
 \ifnum\count0>0
 \divide\count2 by 10
 \advance\count0 by \m@ne
\repeat
```

Here is the recursive part,

```
\ifnum\count2>0 \g@ldens{#1}\fi
```

and then reconstruct the rest of the number.

```
\count0=#1
\loop
 \ifnum\count0>0
 \multiply\count2 by 10
 \advance\count0 by\m@ne
\repeat
\count2=-\count2
\advance\count2 by \count3 \du@zendprint{#1}}}
```

`\du@zendprint` The macro `\du@zendprint` takes care for correctly printing the separator and possible trailing zeros.

```
\def\du@zendprint#1{\ifwiths@p\sep@rator\zerop@d{#1}{\number\count2}%
 \else\zerop@d{1}{\number\count2}\fi \global\withs@ptrue}
```

`\zerop@d` The macro `\zerop@d` uses at least `#1` digits for printing the number `#2`, padding with zeros when necessary. Initially this was a nice macro using tail recursion until it was found that the running time of that macro was proportional to n^2, where n is roughly the number of zeros to be padded. The worst case situation is when printing "0". The running time of the current version is only linear in n.

It is done within an extra pair of braces, so that `\count0` and `\count1` can be used without disturbing their values in other macros.

```
\def\zerop@d#1#2{{\count0=1 \count1=#2
```

First determine the number of digits of `#2` (expressed in the decimal system). This number is in `\count0` and is at least 1.

```
\loop
 \divide \count1 by 10
 \ifnum\count1>0 \advance\count0 by\@ne
\repeat
```

The number of zeros to be padded is max(0, `#1`-`\count0`) (the second argument can be negative), so a simple loop suffices.

```
\loop
 \ifnum\count0<#1\relax 0\advance\count0 by\@ne
\repeat
\number#2}}
```

`\zetg@ld` This macro takes care for several things: it increases the total for a given F-column, it records the largest width of the entries in that column and it typesets `#1` via `\geldm@cro`.

```
\def\zetg@ld#1#2{\global\advance\csname
FCtot@\romannumeral\FCsc@l\endcsname by #1
\setbox0=\hbox{$\geldm@cro{#1}{#2}$}%
\ifdim\wd0>\csname FCwd@\romannumeral\FCsc@l\endcsname
 \global\csname FCwd@\romannumeral\FCsc@l\endcsname=\wd0
\fi\unhbox0}
```

`\b@fi` The macro `\b@fi` provides the beginning of the financial column. It will be inserted in the column to capture the number entered by the user. The `\let` is only local to the column and is necessary because `\ignorespaces` is automatically inserted by the preamble-generating macro `\@mkpream`. Its effect should be annihilated, otherwise the assignment to `\bedr@g` goes wrong. The plain LaTeX command `\@empty` is used for that. The separator and decimal mark are within a math environment, so you can indeed specify `\,` instead of `\thinspace`, but there is an extra brace around, so it doesn't affect the spacing between the digits (trick copied from `dcolumn`, Ref. [3]).

```
\newcount\bedr@g
\def\b@fi#1#2#3{\def\sep@rator{{#1}}\def\decim@lmark{{#2}}%
\def\sp@l{#3}\let\ignorespaces=\@empty \let\unskip=\@empty
\global\advance\FCsc@l by \@ne
```

The value specified by the user is then captured by `\bedr@g` and this is done in a special way: `\bedr@g` is assigned globally within `\box0`. Why? To later check the width of this box, see `\e@fi`.

```
\setbox0=\hbox\bgroup\global\bedr@g=}
```

`\e@fi` If the user enters a number, say 10, LaTeX will read `\setbox 0 = \hbox \bgroup \global \bedr@g = 10 123 \relax \egroup`, which is perfectly valid. The count `\bedr@g` is 10 now and `\box0` is non-empty, in fact containing the number 123 (although any number would be good: it is discarded anyway). If however nothing was entered by the user, LaTeX will read `\setbox 0 = \hbox \bgroup \global \bedr@g = 123 \relax \egroup`, which is also perfectly valid, but now `\bedr@g` is 123 and `\box0` is empty. So the width of `\box0` can be used to discriminate without actually using its contents in the horizontal list. Later, with the code for `\leeg` a related trick will be used. The space at the beginning of `\e@fi` is important, otherwise the 123 might be concatenated with the user entry, leading to wrong numbers.

```
\def\e@fi{ 123\relax\egroup\ifdim\wd0>\z@
 \zetg@ld{\number\bedr@g}{\sp@l}%
\fi}
```

Please note that this trick does not work for an empty F-column that is ended by `\\`. The reason is that the first thing `\\` does—even before providing the `\cr` —is to issue a `\relax`. This stops the assignment to `\bedr@g` and causes a TeX error, complaining about a missing number. This is then repaired by TeX by inserting a zero, so it doesn't affect the total of that column, but in particular this zero is then typeset, which it is not supposed to. I decided to leave it this way because in practice this is never a problem: simply discard the last `&` and the column before the offending column will now be ended by `\\`. Iterate for this line until a filled F-column or a non-F-column is encountered. This is assuming that you did not specify `|`s in the tabular description: for them to work for the whole table you should have all `&`s in place (this is true in general, not only for `fcolumn`). On the other hand you should never, ever use vertical rules in tables, according to Ref. [2]. And if you really must, then fill the empty F-columns with `\leeg{}`.

Here are adaptations to existing macros.

`\@array` The definition of `\@array` had to be extended slightly because it should also include `\@mksumline` (acting on the same `#2` as `\@mkpream` gets). This change is transparant: it only adds functionality and if you don't use that, you won't notice the difference. It starts by just copying the original definition from the `array` package [1].

```
\def\@array[#1]#2{%
\@tempdima \ht \strutbox
\advance \@tempdima by\extrarowheight
\setbox \@arstrutbox \hbox{\vrule
           \@height \arraystretch \@tempdima
           \@depth \arraystretch \dp \strutbox
           \@width \z@}%
```

Here comes the first change: after each `\\` (or `\cr` for that matter) the ⟨*count*⟩ `\FCsc@l` should be reset. This is easiest done with `\everycr`, but `\everycr` is put to `{}` by `\ialign`, so that definition should change. The resetting should be done globally.

```
\def\ialign{\everycr{\noalign{\global\FCsc@l=0 }}%
 \tabskip\z@skip\halign}
```

Then the definition is picked up again.

```
\begingroup
\@mkpream{#2}%
\xdef\@preamble{\noexpand \ialign \@halignto
\bgroup\@arstrut\@preamble\tabskip\z@\cr}%
\endgroup
```

The combination `\endgroup` followed by `\begingroup` seems redundant, but that is not the case: the `\endgroup` restores everything that was not `\global`. With the following `\begingroup` it is ensured that `\@mksumline` experiences the same settings as `\@mkpream` did.

```
\begingroup
\@mksumline{#2}%
\endgroup
```

As a side product of `\@mksumline` also the ⟨*count*⟩s for the totals and ⟨*dimen*⟩s for the widths of the colums are created. The columns should start fresh, i.e., totals are 0 and widths are 0 pt.

```
\res@tsumline
```

From here on it is just the old definition of `array.sty`.

```
\@arrayleft
\if #1t\vtop \else \if#1b\vbox \else \vcenter \fi \fi
\bgroup
\let \@sharp ##\let \protect \relax
\lineskip \z@
\baselineskip \z@
\m@th
\let\\\@arraycr \let\tabularnewline\\\let\par\@empty \@preamble}
```

Because `\@array` was changed here and it is this version that should be used, `\@@array` should be `\let` equal to `\@array` again.

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

`\@mksumline` The construction of the sumline is much easier than that of the preamble for several reasons. It may be safely assumed that the preamble specifier is grammatically correct because it has already been screened by `\@mkpream`. Furthermore most entries will simply add nothing to `\sumline`, e.g., `@`, `!`, and `|` can be fully ignored. Ampersands are only inserted by `c`, `l`, `r`, `p`, `m`, and `b`. So a specifier like `@{}lflf@{}` will yield the sumline `&\a&&\a\\`, (where `\a` is a macro that prints the desired result of the column, see later). Had the specifier been `l|f||@{    }l|f`, then the same sumline must be constructed: all difficulties are already picked up and solved in the creation of the preamble.

In reality the sumline must be constructed from the expanded form of the specifier, so `@{}lf@{}` will expand as `@{}l>{\b@fi{.}{,}{3,2}}r<{\e@fi}@{}`. The rules for constructing the sumline are now very simple:

- add an ampersand when `c`, `l`, `r`, `p`, `m`, or `b` is found, unless it is the first one (this is the same as in the preamble);
- add a `\a` when `<{\e@fi}` is found;
- ignore everything else;
- close with a `\\`.

(In reality also the column check is inserted just before the `\\`, see `\aut@check`.) To discriminate, a special version of `\testpach` could be written, but that is not necessary: `\testpach` can do all the work, although much of it will be discarded. Here speed is sacrificed for space and this can be afforded because the creation of the sumline is only done once per `\tabular`.

The start is copied from `\@mkpream`.

```
\def\@mksumline#1{\gdef\sumline{}\@lastchclass 4 \@firstamptrue
```

At first the column number is reset and the actual code for what was called `\a` above is made inactive.

```
\global\FCsc@l=0
\let\prr@sult=\relax
```

Then `\@mkpream` is picked up again.

```
\@temptokena{#1}
\@tempswatrue
\@whilesw\if@tempswa\fi{\@tempswafalse\the\NC@list}%
\count0\m@ne
\let\the@toks\relax
```

```
\prepnext@tok
```

Next is the loop over all tokens in the expanded form of the specifier. The change with respect to `\@mkpream` is that the body of the loop is now only dealing with F-classes 0, 2, and 10. What to do in those cases is of course different from what to do when constructing the preamble, so special definitions are created, see below.

```
\expandafter \@tfor \expandafter \@nextchar
 \expandafter :\expandafter =\the\@temptokena \do
 {\@testpach
 \ifcase \@chclass \@classfz
  \or \or \@classfii \or
  \or \or \or \or \or \or \or \@classfx \fi
 \@lastchclass\@chclass}%
```

And the macro is finished by applying the `\aut@check` and appending the `\\` to the sumline. Note that the `\aut@check` is performed *in* the last column, but since it does not put anything in the horizontal list—it only writes to screen and transcript file—, this is harmless.

```
\xdef\sumline{\sumline\noexpand\aut@check\noexpand\\}}
```

`\@addtosumline` Macro `\@addtosumline`, as its name already suggests, adds something to the sumline, like its counterpart `\@addtopreamble` did to the preamble.

```
\def\@addtosumline#1{\xdef\sumline{\sumline #1}}
```

`\@classfx` Class f10 for the sumline creation is a stripped down version of `\@classx`: add an ampersand unless it is the first. It deals with the specifiers `b`, `m`, `p`, `c`, `l`, and `r`.

```
\def\@classfx{\if@firstamp \@firstampfalse \else \@addtosumline &\fi}
```

`\@classfz` Class f0 is applicable for specifiers `c`, `l`, and `r`, and if the arguments of `p`, `m`, or `b` are given. The latter three cases, with `\@chnum` is 0, 1, or 2 should be ignored and the first three cases are now similar to class f10.

```
\def\@classfz{\ifnum\@chnum<\thr@@ \@classfx\fi}
```

`\@classfii` Here comes the nice and nasty part. Class f2 is applicable if a `<` is specified. This is tested by checking `\@lastchclass`, which should be equal to 8. Then it is checked that the argument to `<` is indeed `\e@fi`. This check is rather clumsy but this was the first way, after many attempts, that worked. It is necessary because the usage of `<` is not restricted to `\e@fi`: the user may have specified other LaTeX-code using `<`.

```
\def\@classfii{\ifnum\@lastchclass=8
 \edef\t@stm{\expandafter\string\@nextchar}
 \edef\t@stn{\string\e@fi}
 \ifx\t@stm\t@stn
```

If both tests yield `true` then add the macro to typeset everything.

```
  \@addtosumline{\prr@sult}
```

But we're not done yet: in the following lines of code the appropriate ⟨*count*⟩s and ⟨*dimen*⟩s are created, if necessary. Note that `\FCsc@l` was set to 0 in the beginning of `\@mksumline`, so it is well-defined when `\@classfii` is used.

```
  \global\advance\FCsc@l by \@ne
  \ifnum\FCsc@l>\FCtc@l
```

Apparently the number of requested columns is larger than the currently available number of relevant ⟨*count*⟩s and ⟨*dimen*⟩s, so new ones should be created. What is checked here is merely the existence of `\FCtot@<some romannumeral>`. If it already exists—although it may not even be a ⟨*count*⟩; that is not checked—it is not created by `fcolumn` and a warning is given. In case it is a ⟨*count*⟩ you're just lucky, although any change to this ⟨*count*⟩ is global anyway, so things will be overwritten. In the case it is not a ⟨*count*⟩, things will go haywire and you'll soon find out. The remedy then is to rename your ⟨*count*⟩ prior to `fcolumn` to avoid this name clash.

```
   \expandafter\ifx\csname FCtot@\romannumeral\FCsc@l\endcsname\relax
    \expandafter\newcount\csname FCtot@\romannumeral\FCsc@l\endcsname
   \else
    \message{^^JWarning: FCtot@\romannumeral\FCsc@l \space is already
    defined and it may not even be a <count>. I'll proceed,
    but with fingers crossed. }
   \fi
```

And the same is applicable for the ⟨*dimen*⟩: in case of a name clash you have to rename your ⟨*dimen*⟩ prior to `fcolumn`.

```
   \expandafter\ifx\csname FCwd@\romannumeral\FCsc@l\endcsname\relax
    \expandafter\newdimen\csname FCwd@\romannumeral\FCsc@l\endcsname
```

If the creation was successful, the ⟨*count*⟩ `\FCtc@l` should be increased.

```
    \global\FCtc@l=\FCsc@l
   \else
    \message{^^JWarning: FCwd@\romannumeral\FCsc@l \space is already
    defined and it may not even be a <dimen>. I'll proceed,
    but with fingers crossed. }
   \fi
  \fi
 \fi
\fi}
```

Once created it is not necessary to initialise them here because that is done later in one go.

`\leeg` This macro is used to overrule the default behaviour of the pair `\b@fi` and `\e@fi`. It starts with ending the assignment to `\bedr@g` in the same way that `\e@fi` would normally do. Then the effect of `\e@fi` (that is still in the preamble) is annihilated by `\let`ting it to be `\relax`. This `\let` is only local to the current column. Then the argument to `\leeg` is treated in a similar way as `\e@fi` would do with a typeset number.

Since the user may from time to time also need a column entry other than a number in the table, e.g., `\leeg{p.m.}`, this definition is without at-sign.

```
\def\leeg#1{ 234\relax\egroup \let\e@fi=\relax \setbox0=\hbox{#1}%
\ifdim\wd0>\csname FCwd@\romannumeral\FCsc@l\endcsname
 \global\csname FCwd@\romannumeral\FCsc@l\endcsname=\wd0
\fi\unhbox0}
```

Note that anything may be given as argument to `\leeg`, so in principle it can also be used to cheat: `\leeg{0,03}` will insert `0,03` in the table but it doesn't increase the totals of that column by 3 (assuming `3,2` coding for the separations). But you won't cheat, won't you? It may affect the width, so be careful: don't insert the unabridged version of Romeo and Julia here.

`\prr@sult` The macro `\prr@sult` actually puts the information together. It starts like `\leeg`.

```
\def\prr@sult{ 345\relax\egroup \let\e@fi=\relax
```

Then the information for the last line is computed. It is not sufficient to calculate the width of the result (in points) to use that as the width of the rule separating the individual entries and the result. It may be that the sum is larger (in points) than any of the entries, e.g., when the result of $600 + 600$ is typeset. The width of the rule should be equal to the width of `\hbox{$12{,}00$}` then (using specifier `3,2`). On the other hand the width of the rule when summing 2400 and -2400 should be that of `\hbox{$-24{,}00$}` (or `\hbox{$(24{,}00$}`, see above), not the width of the result `\hbox{$0{,}00$}`. Therefore the maximum of all entry widths, including the result, was calculated.

```
\setbox0=\hbox{$\geldm@cro{\number\csname
FCtot@\romannumeral\FCsc@l\endcsname}{\sp@l}$}%
\ifdim\wd0>\csname FCwd@\romannumeral\FCsc@l\endcsname
 \global\csname FCwd@\romannumeral\FCsc@l\endcsname=\wd0
\fi
\vbox{\hrule width \csname FCwd@\romannumeral\FCsc@l\endcsname
\vskip2pt
\hbox to \csname FCwd@\romannumeral\FCsc@l\endcsname{\hfil\unhbox0}}}
```

`\aut@check` If the number of F-columns is even, it is assumed that they are part of two sets of columns of which each column of the first set should balance the appropriate column of the second set. If on the other hand the number of columns is odd, then at least one column has nothing to balance against and no checking occurs. It is correct to check for oddness of `\FCsc@l` since this `\aut@check` is only performed in the last column of the tabular: the value of `\FCsc@l` now equals the number of columns used in the current tabular (and may differ from `\FCtc@l`).

The output is only to screen and the transcript file; it doesn't change the appearance of your document, so in case the assumption is wrong you can safely ignore the result and go on. The ⟨*count*⟩s 0 and 1 are used here and this can be done because any content of those ⟨*count*⟩s from previous calculations has become irrelevant at this moment.

```
\def\aut@check{\ifodd\FCsc@l\else \count0=\@ne \count1=\FCsc@l
 \divide\count1 by \tw@ \advance\count1 by \@ne
 \loop
  \ifnum\csname FCtot@\romannumeral\count0\endcsname=
   \csname FCtot@\romannumeral\count1\endcsname\else
   \message{^^JWarning: F-columns \number\count0 \space
    and \number\count1 \space do not balance! }%
  \fi
  \ifnum\count1=\FCsc@l\else
   \advance\count0 by\@ne \advance\count1 by\@ne
 \repeat
\fi }
```

\checkfcolumns But the assumptions for `\aut@check` may be wrong, therefore manual control on this checking is also made possible here. The macro `\checkfcolumns` provides a way to the user to check that the appropriate columns are balanced (as it should in a balance). Arguments `#1` and `#2` are the column numbers to compare. It is the responsibility of the user to provide the correct numbers here, otherwise bogus output is generated.

```
\def\checkfcolumns#1#2{\noalign{\ifnum\csname FCtot@\romannumeral#1
\endcsname=\csname FCtot@\romannumeral#2\endcsname\else
 \message{^^JWarning: F-columns #1 and #2 do not balance! }%
\fi}}
```

\res@tsumline Since all changes to the totals and widths of the columns are global, they have to be reset actively at the start of a tabular or array. That is an action by itself, but it may occur more often, on request of the user, therefore a special macro is defined. A side effect of this macro is that `\FCsc@l` is reset to 0. This is an advantage: it should be zero at the beginning of a line in the table (for other lines this is done by the \\).

```
\def\res@tsumline{\FCsc@l=\FCtc@l\loop\ifnum\FCsc@l>0
 \global\csname FCtot@\romannumeral\FCsc@l\endcsname=0
 \global\csname FCwd@\romannumeral\FCsc@l\endcsname=\z@
 \advance\FCsc@l by \m@ne
\repeat}
```

\resetsumline To reset a sumline within a table, it should be done within a `\noalign`.

```
\def\resetsumline{\noalign{\res@tsumline}}
```

That's it!

Acknowledgement

Thanks to Karl Berry for valuable comments regarding the consistency of the installation procedure of this version.

References

[1] Frank Mittelbach and David Carlisle. A new implementation of LaTeX's `tabular` and `array` environment.
[2] Simon Fear. Publication quality tables in LaTeX.
[3] David Carlisle. The `dcolumn` package.

Change History

v0.1
- General: First working version . . . 1

v1.0
- General: Three-argument version is working properly 1

v1.1
- General: Automatic checking of column balance performed when number of F-columns is even (behaviour can be overridden). Empty entries are now recognised and correctly treated as such, except for the one ended by the double backslash. Not serious; workaround possible. Furthermore optimisation of code: minimised the number of private counts and resetting of column counter done in a nicer way. 1

v1.1.1
- General: Installation procedure changed from .ins-in-.dtx to separate .ins and .dtx after discussion with Karl Berry as well as some minor code improvements. 1

Index

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