An Enhanced Implementation of the Allocation Macros in LaTeX 2ε, LaTeX 2.09, and plain-TeX

Bernd Raichle
Stettener Str. 73
D-73732 Esslingen, FRG
Email: `raichle@Informatik.Uni-Stuttgart.DE`

1995/09/24 (v1.1b)

Abstract

This style option reimplements the allocation command macros provided by plain-TeX and adds macros to allow allocation of local registers inside groups.

1 Introduction

1.1 Restrictions of `plain.tex`'s Register Allocation Macros

In `plain.tex`, Donald E. Knuth writes about the macros for register allocation [2, lines 149ff]:

> When a register is used only temporarily, it need not be allocated; grouping can be used, making the value previously in the register return after the close of the group. The main use of these macros is for registers that are defined by one macro and used by others, possibly at different nesting levels. All such registers should be defined through these macros; otherwise conflicts may occur, especially when two or more more macro packages are being used at once.

Additional remarks can be read in the TeXbook [1, p. 346]:

> *Allocation of registers.* The second major part of the `plain.tex` file provides a foundation on which systems of independently developed macros can coexist peacefully without interfering in their usage of registers. The idea is that macro writers should abide by the macro conventions following ground rules: [...] (4) Furthermore, it's possible to use any register in a group, if you ensure that TeX's grouping mechanism will restore the register when you're done with the group, and if you are certain that other macros will not make global assignments to that register when you need it. (5) But when a register is used by several macros, or over long spans of time, it should be allocated by `\newcount`, `\newdimen`, `\newbox`, etc. (6) Similar remarks apply [...] to insertions (which require `\box`, `\count`, `\dimen`, and `\skip` registers all having the same number).

This means, that within plain-TeX and all formats based on plain-TeX a user can not allocate a register for temporary use using the given mechanisms. Nonetheless there is often the need for a allocated register which is only used within a rather small piece of document text and should be set free for reuse in the rest of the formatting run:

If a macro `\macroa` needs a register which should never be destroyed by another macro `\macrob` which is called by `\macroa`, you either have to know about the realisation of `\macrob` or you have to allocate a new register.

The name of this newly allocated register is only known to the macro package in which macro `\macroa` resides. If you have a lot of macros which needs registers by their own it is possible that all of the 256 registers of one type will be allocated. In most cases it would be possible to share some of these registers by more than one macro because the register is only needed within (an environment delimited by) a group. To share registers—if they are used only locally—it will be necessary to provide a mechanism to allocate registers locally. Unfortunately `plain.tex` and all derived formats, like LaTeX, misses this mechanism leading to the problem that some macro packages cannot be combined.

In the following example it is necessary to call two macros `\macroa` and `macrob` in the same group (e.g. because `\macroa` performs special actions at the begin and `\macrob` at the end of an environment). Further you do not have any knowledge about the use of scratch register in `\macroa`, `\macrob` or the text in the argument specified by the user. Therefore it is impossible to use a scratch register to save the current value of `\hsize`; we have to allocate a new register to save it.

```
\newdimen\savedhsize

\def\callAandB#1{\begingroup
    \savedhsize=\hsize
    \hsize=0.8\hsize    \macroa
    #1\relax
    \hsize=\savedhsize  \macrob
  \endgroup}
```

If there will be a mechanism, e.g. by a macro `\lnewdimen`, to allocate a local register it is possible to use this local register inside the group without worrying about possible dependencies between the macros and the user specified text. Additionally after leaving the group, the dimen register will be free to be used by other macros.

```
\def\callAandB#1{\begingroup
    \lnewdimen\savedhsize  \savedhsize=\hsize
    \hsize=0.8\hsize    \macroa
    #1\relax
    \hsize=\savedhsize  \macrob
  \endgroup}
```

1.2 Local Register Allocation Macros

`\newcount`
`\newdimen`
`\newskip`
`\newbox`
`\newinsert`

When using this reimplementation of the allocation macros, the `plain.tex` macros `\newcount`, `\newdimen`, `\newskip`, and `\newbox` will have a changed behaviour in two aspects:

- Do not assume that registers which are allocated consecutively have consecutive register numbers. In general you should not make any assumption about the order of register numbers.
- It is possible that a register was locally allocated and used in the past. After leaving the group the register is set to free and the old value—normally zero—is restored. Nonetheless you should not assume that a newly allocated register is initialised with the value zero.

`\newmuskip` `\newtoks` The behaviour of the macros `\newmuskip` and `\newtoks` are not changed because they are not used for insertions. Nonetheless this can be changed in a future version.

`\lnewcount` `\lnewdimen` `\lnewskip` `\lnewmuskip` `\lnewbox` `\lnewtoks` `\lnewcount`, `\lnewdimen`, `\lnewskip`, `\lnewmuskip`, `\lnewbox`, and `\lnewtoks` are the set of new macros to allocate registers within the scope of the current group. At the end of the current group the allocated register is freed.

As with their global counterparts you should not assume that the registers are allocated in a consecutive order—neither increasing nor decreasing—or that the allocated register is initialised with zero.

1.3 Usage Examples

1.3.1 LaTeX's Picture Environment

LaTeX's picture environment uses the box register `\@picbox` and the dimen register `\@picht` to save the contents of a picture and the user specified height of this picture at the beginning of the environment. These two registers are set at the beginning and used at the end of the `picture` environment. Because the two registers are used nowhere else, it is possible to share them by only allocating them within the environment.

This is the simplified code for the `\begin{picture}` and `\end{picture}` commands; the macros `\begin` and `\end` start and end a new group:

```
\newbox\@picbox
\newdimen\@picht

\def\picture(#1,#2){%
  \@picht = #2\unitlength  % save picture height
  \setbox\@picbox = \hbox to #1\unitlength\bgroup
    \ignorespaces}

\def\endpicture{%
    \hss
  \egroup                   % close picture box
  \ht\@picbox = \@picht     % set box height to save picture height
  \dp\@picbox = 0pt         % set box depth to 0pt
  \mbox{\box\@picbox}}      % output this box
```

Using the new local allocation macros the code changes is simple: Move the `\newbox` and `\newdimen` command within the `\picture` macro definition and rename them to `\lnewbox` and `\lnewdimen`:

```
%\newbox\@picbox
%\newdimen\@picht
```

```
\def\picture(#1,#2){%
  \lnewbox\@picbox
  \lnewdimen\@picht
  \@picht = #2\unitlength  % save picture height
  \setbox\@picbox = \hbox to #1\unitlength\bgroup
    \ignorespaces}
```

Additionally we can use a property of the usage pattern of these two registers: Within a nested use of the `picture` environment—you can use subpictures within a picture—these registers are allocated for each nesting level. This is not really necessary because the box and the dimen register is always used in a group and is never changed globally. It is possible to add tests to avoid a (local) reallocation of these registers:

```
\let\@picbox=\relax    % mark register as not yet allocated
\let\@picht =\relax    % mark register as not yet allocated

\def\picture(#1,#2){%
  \ifx\@picbox\relax \lnewbox\@picbox  \fi
  \ifx\@picht\relax  \lnewdimen\@picht \fi
  \@picht = #2\unitlength  % save picture height
  \setbox\@picbox = \hbox to #1\unitlength\bgroup
    \ignorespaces}
```

1.3.2 LaTeX's `\multiput` Macro

Another example of registers which are used only in one rarely used macro, are the count register `\@multicnt` and the two dimen registers `\@xdim` and `\@ydim` for the `\multiput` command. To change it according to the scheme above you have to insert an additional grouping.

In LaTeX 2ε these three rarely used registers and some more are shared with the `\bezier` command, whose definition has to be changed accordingly. This shows the need—and advantage—of a better allocation scheme.

1.3.3 PICTeX

The macro package PICTeX uses a lot of dimen registers. A subset of these registers are only used within a picture or within a set of drawing and parameter setting commands.

As an example the dimen registers `\totalarclength` is only used in a very restrictic set of commands, i.e., it can be allocated within the scope of the higher level commands or within PICTeX's `\beginpicture` … `\endpicture` environment.

2 Implementation

2.1 Prolog, LaTeX 2ε Package Declarations, etc.

The very first thing we do is to ensure that this file is not read in twice. To this end we check whether the macro `\lnewcount` is defined. If so, we just stop reading this file.

```
⟨+package⟩\expandafter\ifx\csname lnewcount\endcsname\relax
⟨+package⟩\else \expandafter\endinput \fi
```

For the sake of LaTeX 2ε users declarations for this package are included. The additional group hack is needed to prevent the definition of `\ProvidesPackage` as `\relax` if it is undefined.

```
⟨+package | test⟩\begingroup\expandafter\expandafter\expandafter\endgroup
⟨+package | test⟩\expandafter\ifx\csname ProvidesPackage\endcsname\relax
⟨+package | test⟩\else
  \NeedsTeXFormat{LaTeX2e}
⟨+package⟩  \ProvidesPackage{localloc}%
⟨+test⟩  \ProvidesFile{localtst.tex}%
⟨*!package&!test⟩
  \ProvidesFile{localloc.dtx}%
⟨/!package&!test⟩
     [1995/09/24 v1.1b %
      Additional LaTeX Kernel Macros (local allocation)]
⟨+package | test⟩\fi
```

```
⟨*package⟩
```

To allow the use of this file within a plain-TeX job, we have to change the category code of the ‘at’ character `@`.

```
\expandafter\chardef\csname lcllc@at\endcsname=\catcode`\@
\catcode`\@=11 % \makeatletter
```

2.2 Accessing the `muskip` registers

Beside the well-known TeX registers, like `\count` or `\box`, there is another set of registers which are used only in few macros. The `\muskip` registers are normally used in math mode only and the coercion of the value in a `\muskip` register to an integer (`\count`), dimension (`\dimen`), or glue (`\glue`) produces an error.

The unit of a `\muskip` value is 1 mu—there are 18 mu in 1 em of the current font in *math family 2*. If you assign a value to a `\muskip` register, this value is represented in the unit mu, not in pt. The dimension of the `\muskip` value is computed, if this value is used by `\mskip` or `\mkern`.

To use a `\muskip` register as a replacement for a `\count` register, it is necessary to do assignments, additions, and other operations in units of mu. If you want to coerce an integer to a `\muskip` register, you have to get the value with `\the` and add the keyword `mu` while scanning the number. To get the integer value of a `\muskip` register, you can use the following macro `\lcllc@cnvmutocnt`.

`\lcllc@cnvmutocnt` The macro `\lcllc@cnvmutocnt` ist called with a `\muskip` register or a value in `mu` units as its argument. It will expand to the integer value. To get the value of a register, we first expand the value of this register to a list of tokens with TeX’s `\the` primitive. Example: if the register has the value 3.4 mu, the expansion of `\the` produces the tokens $\texttt{3}_{12}\texttt{.}_{12}\texttt{4}_{12}\texttt{m}_{12}\texttt{u}_{12}$.

```
\def\lcllc@cnvmutocnt#1{\expandafter\lcllc@delmukey\the#1}
```

After we have expanded the value of the register, it is necessary to strip of the unit keyword `mu` and the fraction of the value. To do this, we have to use tokens with *category code* 12 to match the expansion string produced by `\the`.

```
\begingroup \catcode`\M=12 \catcode`\U=12 \catcode`\.=12
  \lowercase{\endgroup
\def\lcllc@delmukey#1.#2MU{#1}}
```

2.3 Allocation registers

2.3.1 Registers defined in `plain.tex`

Plain-TeX's allocation macros use a fixed set of `count` registers containing a set of numbers with the highest allocated register number. To make the following macro code more readable, alias names for these `count` registers are defined.

```
\countdef\count@ptr    = 10
\countdef\dimen@ptr    = 11
\countdef\skip@ptr     = 12
\countdef\muskip@ptr   = 13
\countdef\box@ptr      = 14
\countdef\toks@ptr     = 15
\countdef\read@ptr     = 16
\countdef\write@ptr    = 17
\countdef\mathfam@ptr  = 18
```

`plain.tex` for TeX 3 introduces a new allocation command needing an additional allocation count register. To simplify the code of this package with older versions of `plain.tex`, a dummy count register is allocated and used.

```
\begingroup\expandafter\expandafter\expandafter\endgroup
\expandafter\ifx\csname newlanguage\endcsname\relax
  \csname newcount\endcsname\language@ptr
\else
  \countdef\language@ptr = 19
\fi
```

The allocation command for insertion uses another register, which is already aliased as `\insc@unt`.

```
\countdef\insc@unt = 20 % = 19 in older versions of ‘plain.tex’
```

2.3.2 Additional Registers

For the new local allocation macros additional registers are needed containing the boundaries for all types of locally allocated registers. Whereas all globally allocated registers are located in the lower region of the register numbers (with the exception of the set of registers needed for an insertion), locally allocated registers are taken from the higher region of the register numbers—from top towards the bottom.

To avoid using six additional count registers we use `\muskip` registers to save the current lower boundary of the allocatable register region. These boundaries are initialised either with 256 or the actual boundary of the insertion allocation register.

```
\newmuskip\count@limit  \count@limit  = \the\insc@unt mu
\newmuskip\dimen@limit  \dimen@limit  = \the\insc@unt mu
\newmuskip\skip@limit   \skip@limit   = \the\insc@unt mu
\newmuskip\box@limit    \box@limit    = \the\insc@unt mu
\newmuskip\muskip@limit \muskip@limit = 256mu
\newmuskip\toks@limit   \toks@limit   = 256mu
```

To simplify the code of the following allocation macros and to save macro space, two additional `\muskip` registers are allocated and initialised with the constants 16 and 256.

```
\newmuskip\cclvi@mu      \cclvi@mu      = 256mu
\newmuskip\xvi@mu        \xvi@mu        = 16mu
```

2.4 Allocation Macros for the “End User”

2.4.1 Global Allocation Macros

`\newcount` `\newdimen` `\newskip` `\newmuskip` `\newbox` `\newtoks` `\newread` `\newwrite` `\newfam` `\newlanguage`

We have to replace the original macros in `plain.tex` by new ones to allow locally allocated registers. Otherwise it will be possible that the two regions, the lower one with globally allocated registers and the higher one with locally allocated registers, will overlap resulting in a double use of the same register.

With the new macros it will be unnecessary to un-outer the `plain.tex` macros, because the local allocation macros have to be non outer. Nonetheless we want to use this package in conjunction with LaTeX and therefore we have to define some of them as non outer.

The top level macros use the low level macro `\alloc@` which has the following parameters: The boundaries of the lower and the higher region, the type of the allocated register, a flag if we have to check on overlapping with insertion registers, the TeX primitive to be used to define this register, and the user given control sequence.

```
%\outer
\def\newcount  {\alloc@\count@ptr  \count@limit \count  1\countdef }
%\outer
\def\newdimen  {\alloc@\dimen@ptr  \dimen@limit \dimen  1\dimendef }
%\outer
\def\newskip   {\alloc@\skip@ptr   \skip@limit  \skip   1\skipdef  }
% \outer
\def\newmuskip {\alloc@\muskip@ptr \muskip@limit\muskip 0\muskipdef}
%\outer
\def\newbox    {\alloc@\box@ptr    \box@limit   \box    1\chardef  }
% \outer
\def\newtoks   {\alloc@\toks@ptr   \toks@limit  \toks   0\toksdef  }
% \outer
\def\newread   {\alloc@\read@ptr   \xvi@mu      \read   0\chardef  }
%\outer
\def\newwrite  {\alloc@\write@ptr  \xvi@mu      \write  0\chardef  }
%\outer
\def\newfam    {\alloc@\mathfam@ptr\xvi@mu      \fam    0\chardef  }
% \outer
\def\newlanguage{\alloc@\lang@ptr  \cclvi@mu   \language0\chardef  }
```

`\newinsert` needs additional code because this macro has to allocate a set of `\box`, `\count`, `\dimen`, and `\skip` registers with the restriction that all registers have to have the same number. `\newinsert` is redefined at the end of this file.

2.4.2 Local Allocation Macros

`\lnewcount` `\lnewdimen` `\lnewskip` `\lnewmuskip` `\lnewbox` `\lnewtoks`

There is an equivalent set of macros to allocate registers locally. Instead of using the low level macro `\alloc@` it uses the macro `\lalloc@` with the same set of parameters.

```
\def\lnewcount {\lalloc@\count@ptr \count@limit \count  1\countdef }
\def\lnewdimen {\lalloc@\dimen@ptr \dimen@limit \dimen  1\dimendef }
```

```
\def\lnewskip  {\lalloc@\skip@ptr  \skip@limit  \skip   1\skipdef  }
\def\lnewmuskip{\lalloc@\muskip@ptr\muskip@limit\muskip 0\muskipdef}
\def\lnewbox   {\lalloc@\box@ptr   \box@limit   \box    1\chardef  }
\def\lnewtoks  {\lalloc@\toks@ptr  \toks@limit  \toks   0\toksdef  }
```

Local allocation macros for `\read`, `\write` streams, math `\fam`, `language` numbers, or insertions are possible but not very useful because of the global nature of these resources.

2.5 Low Level Allocation Macros

The allocation scheme used by the macros in `plain.tex` is simple:

- all allocations are global,
- register, math family, language, input/output stream numbers are allocated using the lowest possible number, i.e., it is done from lower number to higher numbers,
- the set of insertion registers (`box`, `count`, `skip`, and `dimen`), which have to have the same register number, are allocated beginning from 255 to lower numbers.

Contrary to this simple scheme this package uses numbers for locally allocated registers from top to bottom interfering with the original insertion register allocation scheme *and* the simple test for a register number overflow.

`\lcllc@insnums`

Insertion registers are now allocated in a non-monotonous order using the lowest possible number (because the allocation of the needed set of registers is done in a global way). To allow the checking of overlapping of globally or locally allocated registers with the registers used for insertions, the insertion register numbers are saved as a list in `\lcllc@insnums`. Each entry in this list is saved as `\do`$\langle n \rangle$`.`; the list is initialised as empty.

```
\def\lcllc@insnums{}
```

`\lcllc@checkins`

Whenever a new `box`, `count`, `skip`, or `dimen` register is allocated it has to be checked first, if the next possible free register is already allocated for an insertion.

To detect a collision between a possibly free register number and the set of insertion registers, the macro `\lcllc@checkins` is called with the register number to be tested in `\allocationnumber`. The argument of this macro has to be either 1 or -1 and is used to get the next register number to be tested against the insertion number list if an additional iteration is needed. The result is a register number in `\allocationnumber` which is not used for an insertion. (Note: Nonetheless it can be an out-of-region register number of an already used register!)

```
\def\lcllc@checkins#1{%
  \begingroup
```

After opening a new group, the macro `\do` is defined to read the next insertion number delimited by a simple dot. It then tests this number against the current `\allocationnumber` and if the register numbers are equal, `\allocationnumber` is advanced by the given argument (which is 1 or -1).

To avoid a global change of `\allocationnumber`, a group hack with expansion is used: If the list `\lcllc@insnums` is expanded with this definition of `\do`, the result is empty if the register number is not used for an insertion. Otherwise it expands to the `\advance` command followed by an additional `lcllc@checkins` (which has to be protected by `\noexpand`) to start another iteration with the changed `\allocationnumber`.

```
    \def\do##1.{%
      \ifnum##1=\allocationnumber
        \advance\allocationnumber#1\relax
        \noexpand\lcllc@checkins{#1}%
      \fi}%
```

The following expanded definition text of `\x` contains code to close the group and the expansion of `\lcllc@insnums`, which is either empty or contains the change of `\allocationnumber` and an iterative call of `\lcllc@checkins`. After the definition is completed, the code is executed which closes the group.

```
    \edef\x{\endgroup \lcllc@insnums}%
  \x}
```

`\alloc@`

`\alloc@` is the low level macro for global allocation, which is called with six arguments. The macro itself only uses the first four argument: The two boundaries, the item type to be allocated, and a flag if this item is propably used for an insertion. The remaining two arguments, the TEX primitive and the control sequence to assign the allocated number is used by `\lcllc@def`.

```
\def\alloc@#1#2#3#4{%
```

In the first step we assign the boundary of the lower region for this item type to `\allocationnumber` and increment it by one.

```
  \allocationnumber#1%
  \advance\allocationnumber\@ne
```

If this item type is used by insertion, we have to check for collision with the `\lcllc@checkins` macro. After the execution of this macro, `\allocationnumber` contains the next free number to be used for allocation. This number is assigned globally to the register containing the boundary.

```
  \ifnum#4=\@ne \lcllc@checkins\@ne \fi
  \global#1\allocationnumber
```

We now have found an unused register. Nonetheless it is possible that the register number is either not in the range of existing register number or it is used for a locally allocated register. `\ch@ck` is used to test these things.

```
  \ch@ck#1#2#3%
```

At the end the ⟨*control sequence*⟩ is defined using the found `\allocationnumber`.

```
  \lcllc@def\global{}#3}
```

`\lalloc@`

`\lalloc@` is the low level macro for local allocation. The definition of this macro a copy of `\alloc@` with some small changes.

```
\def\lalloc@#1#2#3#4{%
```

In the first step we assign the boundary of the *higher* region and *decrement* it by one. Because the higher boundary is saved in a `\muskip` register, we have to use `\lcllc@cnvmutocnt` to get the boundary value.

```
  \allocationnumber\lcllc@cnvmutocnt#2%
  \advance\allocationnumber\m@ne
```

The check for insertion number collisions is called with -1 to decrement the register number to get the next free register number. The `\allocationnumber` is then assigned *locally* to the `\muskip` register containing the boundary.

```
  \ifnum#4=\@ne \lcllc@checkins\m@ne \fi
  #2\the\allocationnumber mu %
```

The check for an already used globally allocated register or a range underflow is needed, too.

```
  \ch@ck#1#2#3%
```

At the end the ⟨*control sequence*⟩ is defined using the found `\allocationnumber`.

```
  \lcllc@def\relax{(local)}#3}
```

`\ch@ck` The test macro `\ch@ck` for a collision of the globally and locally allocated registers from `plain.tex` has to be changed because the higher boundary is given as a `\muskip` value instead of a normal integer.

```
\def\ch@ck#1#2#3{%
  \ifnum#1<\lcllc@cnvmutocnt#2\relax \else
    \errmessage{No room for a new #3}%
    % \allocationnumber\m@ne
  \fi}
```

`\lcllc@def` `\lcllc@def` is used to assign the found `\allocationnumber` to the ⟨*control sequence*⟩ using the TeX primitive. Additionally an entry is written to the `log` file.

```
\def\lcllc@def#1#2#3#4#5{%
  #1#4#5=\allocationnumber
  \wlog{\string#5#2=\string#3\the\allocationnumber}}
```

2.5.1 Low Level Insertion Allocation Macros

Insertions need a set of registers with the same register number. To find a new register number when allocating an insertion, we have to get the maximum of all boundaries used for an insertion.

`\lcllc@getmax` `\lcllc@getmax` is used to assign `\allocationnumber` to the maximum of `\allocationnumber` and its argument.

```
\def\lcllc@getmax#1{%
  \ifnum#1<\allocationnumber \else
    \allocationnumber#1\advance\allocationnumber\@ne
  \fi}
```

`\newinsert` `\newinsert` globally allocates a set of registers used for an insertion.

```
\outer\def\newinsert{%
```

First we have to find the maximum of all globally allocated register number used for an insertion. We start by setting `\allocationnumber` to -1...

```
  \allocationnumber\m@ne
```

... and get the maxmimum of the boundaries for `count`, `dimen`, `skip`, and `box` registers.

```
  \lcllc@getmax\count@ptr \lcllc@getmax\dimen@ptr
  \lcllc@getmax\skip@ptr  \lcllc@getmax\box@ptr
```

Then we have to check if this `\allocationnumber` is already used for another insertion.

```
  \lcllc@checkins\@ne
```

At last we have to check if this `\allocationnumber` is used for one of the locally allocated registers.

```
  \ch@ck\allocationnumber\count@limit\count
  \ch@ck\allocationnumber\dimen@limit\dimen
  \ch@ck\allocationnumber\skip@limit\skip
  \ch@ck\allocationnumber\box@limit\box
```

The found register number for insertions is then (globally) inserted into the list of insertion numbers.

```
  \begingroup
    \let\do\relax
    \xdef\lcllc@insnums{\do\the\allocationnumber.\lcllc@insnums}%
  \endgroup
```

At the end the ⟨*control sequence*⟩ is defined using the found `\allocationnumber`.

```
  \lcllc@def\global{}\insert\chardef}
```

The register `\insc@unt` which is used to contain the boundary number for insertion registers in the original `plain.tex` macro is not used anymore.

Finally, the category code of the ‘at’ character `@` is reset to its original value.

```
\catcode`\@=\lcllc@at
⟨/package⟩
```

References

[1] Donald E. Knuth, *The TEXbook*, Addison-Wesley Publ., Reading, Mass., Juni 1991.

[2] File `plain.tex`, version 3.14159, March 1995.