Documentation for multido.tex: A loop macro for Generic TEX

Timothy Van Zandt*

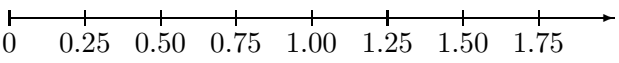
Version 1.0
28 September 1992

1 Usage notes

`multido.tex`/`multido.sty` contains the `\multido` macro, which is a loop facility for Generic TeX. This macro happens to be useful for drawing pictures, and was originally developed for the PSTricks package,[1] but you can use it for other purposes as well.

A special feature is support of fixed-point addition. For example, PSTricks uses the `\multido` to put numbers on axes, much like in the following LATEX example: PSTricks uses `\multdo` internally to put numbers on axes, much like in this LATEX example:

```
\setlength{\unitlength}{1cm}
\small
\begin{picture}(8,1)(0,-.5)
\put(0,0){\vector(1,0){8}}
\multido{\i=0+1,\n=0+0.25}{8}{%
  \put(\i,-.1){\line(0,1){.2}}
  \put(\i,-.2){\makebox[t](0,0){\n}}}
\end{picture}
```



The general syntax for `\multido` is:

`\multido`{*variables*}{*repetitions*}{*stuff*}

*Author's address: Department of Economics, Princeton University, Princeton, NJ 08544-1021, USA. Internet: `tvz@Princeton.EDU`

[1] PSTricks is an extensive collection of PostScript-based macros for Generic TeX. It is available from the /pub directory at Princeton.EDU, and TEX archives.

stuff is whatever you want repeated; it can be any balanced TEX input. *repetitions* is the number times *stuff* is repeated.

The first argument is the interesting one. *variables* is a comma-separated list of variable declarations.[2] Each variable declaration is of the form:

variable = *initial value* + *increment*

variable is a command sequence that can be used in *stuff*. It is initially set to *initial value*, and is then incremented by *increment* with each repetition.

The first letter of the variable name determines the variable type. There are four variable types:

Dimension (d or D) The initial value and the increment should be dimensions (lengths, in LATEX parlance). The substitution text is a dimension, with `sp` units. E.g., `\dx=4cm+5pt`.

Number (n or N) The initial value and increment should be integers or numbers with the same number of digits to the right of the decimal. The one exception is that it is always OK for the initial value to be an integer. There can be at most 8 digits on each side of the decimal. The substitution text is a number, with fixed-point addition. E.g., `\n=3+7.05`, `\Nx=5.30+-1.25`.

Integer (i or I) The initial value and increment should be integers. This gives the same result as using a number variable, but it is faster. E.g., `\I=2+-1`.

Real (r or R) The initial value and increment should be integers or numbers with at most 4 digits on each side of the decimal. The substitution text is a number, but with floating point addition and occasional small errors. This gives a less satisfactory result than using a number variable, but it is faster. E.g., `\ry=4.2+1.05`.

Here are some examples that illustrate how the substitution text is determined:

```
\multido{}{10}{\TeX\ }
```
TEX TEX TEX TEX TEX TEX TEX TEX TEX TEX

```
\multido{\d=2pt+3pt}{5}{\d, }
```
131072sp, 327680sp, 524288sp, 720896sp, 917504sp,

```
\multido{\n=2+3}{10}{\n, }
```
2, 5, 8, 11, 14, 17, 20, 23, 26, 29,

```
\multido{\i=2+3}{10}{\i, }
```
2, -1, -4, -7, -10, -13, -16, -19, -22, -25,

[2] Don't use commas to mark the decimal point within the *variables* argument, as they will be confused for delimiters.

`\multido{\r=2+3.05}{6}{\r, }`

2.0, 5.05, 8.1, 11.15001, 14.20001, 17.25002,

`\multido{\n=2.00+3.05}{8}{\n, }`

2.00, -1.05, -4.10, -7.15, -10.20, -13.25, -16.30, -19.35,

Here are some details about the choice of names:

- Your computer won't explode if you use names that conflict with TEX internal commands, but you might want to check name conflicts if you get inexplicable errors. The command `\MultidoCheckNames` can be useful in this case. It causes `\multido` to report an error whenever you use a variable name that is already defined. But see the next item.
- The whole `\multido` loop is grouped. This means, e.g., that although `\i` is a Plain TEX command sequence (giving a dotless "ı"), you can use the variable `\i` if you do not use any dotless i's in *stuff* (and if you do not use `\MultidoCheckNames`).

Here are a few more details:

- `\Multido` commands can be nested.
- Spaces after a `\multido` command are ignored. This makes `\multido` more hospitable for pictures.
- Spaces between the various parts of the *variables* argument are ignored.

And finally here a few special features, some of which are of interest mainly macro writers and other TEXnicians:

- The material that is repeated is not grouped, so that you can insert your own recursive routines.
- There is a variant, `\mmultido`, which works just like `\multido` except that the variables are all incremented once before starting.
- There are variants, `\Multido` and `\MMultido` of `\multido` and `\mmultido`, resp., that do not group the whole loop. This can be useful, e.g., for making entries in an alignment environment. However, these cannot be nested within any `\multido` macro.
- If the number of repetitions is a negative number, the variables are incremented backwards.
- The count register `\multidocount` keeps track of the number of the iterations.

- The command `\multidostop` causes the `\multido` loop to quit at the end of the current iteration.

- Fixed point addition is performed by `\FPadd` and `\FPsub`:

  `\FPadd`{*num1*}{*num2*}{*cs*}
  `\FPsub`{*num1*}{*num2*}{*cs*}

  num2 is added to or subtracted from *num1*, and the answers is stored in the command sequence given as the third argument. The rules about decimals and so on that apply to number variables apply here as well. E.g., after

  ```
  \FPsub{1.75}{-0.15}{\answer}
  ```

  the definition of `\answer` is 1.90.

2 Documented code

Yes, the documentation is pretty scimpy.

These macros use TEX primitives, plus the Plain TEX commands:

> `\dimen@`, `\dimen@i`, `\dimen@ii`, `\count@`, `\newcount`, `\newtoks`, `\@ne`, `\tw@`, and `\@z@`.

Check whether file has been loaded already.

```
\expandafter\ifx\csname multido@\endcsname\relax\else
  \expandafter\endinput
\fi
```

Identify the file on the terminal:

```
\message{\space\space v\fileversion\space\space
  \filedate\space\space <tvz}
```

Take care of the catcode of `@`:

```
\edef\theatcode{\the\catcode`\@}
\catcode`\@=11
```

There are some hacks borrowed from PSTricks, which are loaded if PSTricks is not being used. `\@dimtonum` strips the value of `#1`, a dimension register, of the `pt`, and assigns the result to `#2`, a command sequence.

```
\expandafter\ifx\csname @pstrickserr\endcsname\relax
  \def\@empty{}
  \def\@nnil{\@nil}
  \def\@dimtonum#1#2{\edef#2{\@@dimtonum#1}}
  \def\@@dimtonum#1{\expandafter\@@@dimtonum\the#1}
  {\let\@nnil\expandafter\catcode`\p=12\catcode`\t=12
  \global\@nnil\def\csname @@@dimtonum\endcsname#1pt{#1}}
\fi
```

`\multido@count`, `\multidocount`

`\multido@count` stores the number of repetitions. `\multidocount` keeps track of the iteration. These are also used locally as scratch counters by `\FPadd@`.

```
\newcount\multido@count
\newcount\multidocount
```

`\multido@stuff`

`\multido@stuff` is used to store the *stuff* that is to be repeated.

```
\newtoks\multido@stuff
```

\multido, \mmultido, \Multido, \MMultido

```
\def\multido{\multido@{}{\begingroup}{\endgroup}}
\def\mmultido{\multido@{\multido@stepvar}{\begingroup}{\endgroup}}
\def\Multido{\multido@{}{}{}}
\def\MMultido{\multido@{\multido@stepvar}{}{}}
```

\multido@

\multido@initvar processes the variable declarations, initializing the value of the variables and defining \multido@stepvar to increment the variables with each repetition. \multido@count is set to the number of repetitions.

```
\def\multido@#1#2#3#4#5#6{%\
  #2%
    \multido@count=#5\relax
    \def\multido@stepvar{}%
    \def\do{\noexpand\do\noexpand}%
    \multido@initvar#4,\@nil,%
    \let\do\noexpand
    \edef\multido@stepvar{\multido@stepvar}%
    \ifnum\multido@count<\z@\multido@count=-\multido@count\fi
    \multidocount=1\relax
    #1%
    \multido@stuff{#6}%
    \multido@loop
  #3%
  \ignorespaces}
```

\multido@loop

\multido@loop does the repetition.

```
\def\multido@loop{%
  \the\multido@stuff
  \ifnum\multidocount<\multido@count
    \advance\multidocount\@ne
    \multido@stepvar
    \expandafter\multido@loop
  \fi}
```

\multidostop

```
\def\multidostop{\multidocount=\multido@count}
```

\multido@initvar, \multido@@initvar, \multido@getvartype

\multido@initvar passes each variable declaration to \multido@@initvar.

```
\def\multido@badvar{%
  \multido@count=0
  \errhelp{\multido command will be skipped.}%
  \errmessage{Bad \string\multido\space variable declaration}}
\def\multido@initvar#1,{%
  \def\multido@temp{#1}%
  \ifx\multido@temp\@nnil\else
    \ifx\multido@temp\@empty\else
      \multido@@initvar#1\@nil=+\@nil\relax
    \fi
    \expandafter\multido@initvar
  \fi}
\def\multido@@initvar#1=#2+#3\@nil#4\relax{%
  \ifx\@empty#4\@empty
    \multido@badvar
  \else
    \multido@vartype#1\@empty
    \ifx\multido@temp\relax
      \multido@badvar
    \else
      \multido@temp{#2}{#3}#1%
    \fi
  \fi}
\def\multido@vartype#1{%
  \ifcat\noexpand\@nil\noexpand#1%
     \expandafter\multido@@vartype\string#1\@nil
  \else
    \let\multido@temp\relax
  \fi}
\def\multido@@vartype#1#2#3\@nil{%
  \expandafter\let\expandafter\multido@temp
    \csname multido@init@#2\endcsname}
```

\MultidoCheckNames

```
\def\MultidoCheckNames{%
  \let\multido@@@vartype\multido@vartype
  \def\multido@vartype##1{%
    \ifx\undefined##1%
      \multido@@@vartype{##1}%
    \else
      \errhelp{\multido command will be skipped.}%
      \errmessage{Multido variable \string##1 already defined}%
      \let\multido@temp\relax
```

```
87     \fi}}
```

For each variable type, we must now define `\multido@init`*vartype*. The syntax of these macros is:

`\multido@init@`*vartype*{*initial value*}{*increment*}{*variable*}

and the outcome should be:

1. Set *variable* to the initial value.
2. Invoke `\multido@addtostep`{*step stuff*}, where *step stuff* is whatever should be done to increment the variable. {*step stuff*} is expanded first, with expansion suppressed by `\do`.

`\multido@addtostep`

```
88 \def\multido@addtostep#1{\edef\multido@stepvar{\multido@stepvar#1}}
```

`\multido@init@d`, `\multido@init@D`, `\multido@step@d`

We start with the variable type for dimensions (`d`, `D`).

```
89 \def\multido@init@d#1#2#3{%
90   \dimen@=#1\relax
91   \edef#3{\number\dimen@ sp}%
92   \dimen@=#2\relax
93   \ifnum\multido@count<\z@\dimen@=-\dimen@\fi
94   \multido@addtostep{\do\multido@step@d{\do#3}{\number\dimen@ sp}}}
95 \def\multido@step@d#1#2{%
96   \dimen@=#1\advance\dimen@#2
97   \edef#1{\number\dimen@ sp}}%
98 \def\multido@init@D{\multido@init@d}
```

`\multido@init@i`, `\multido@init@I`, `\multido@step@i`

Now the variable type for integers (`i`, `I`).

```
99 \def\multido@init@i#1#2#3{%
100   \count@=#1\relax
101   \edef#3{\the\count@}%
102   \count@=#2\relax
103   \ifnum\multido@count<\z@\count@=-\count@\fi
104   \multido@addtostep{\do\multido@step@i{\do#3}{\the\count@}}}
105 \def\multido@step@i#1#2{%
106   \count@=#1\advance\count@ by #2
107   \edef#1{\the\count@}}
108 \def\multido@init@I{\multido@init@i}
```

\multido@init@r, \multido@init@R, \multido@step@r

Now the variable type for reals (`r`, `R`).

```
\def\multido@init@r#1#2#3{%
  \dimen@=#1pt
  \@dimtonum\dimen@#3%
  \dimen@=#2pt
  \ifnum\multido@count<\z@\dimen@=-\dimen@\fi
  \multido@addtostep{\do\multido@step@r{\do#3}{\number\dimen@ sp}}}
\def\multido@step@r#1#2{%
  \dimen@=#1pt\advance\dimen@#2
  \@dimtonum\dimen@#1}
\def\multido@init@R{\multido@init@r}
```

\multido@init@n, \multido@step@n

Now the variable type for numbers (`n`, `N`).

```
\def\multido@init@n#1#2#3{%
  \edef#3{#1}%
  \ifnum\multido@count<\z@\expandafter\FPsub\else\expandafter\FPadd\fi
  {0}{#2}\multido@temp
  \multido@addtostep{\do\FPadd{\do#3}{\multido@temp}{\do#3}}}
\def\multido@init@N{\multido@init@n}
```

\FPadd, \FPsub

Simple fixed-point addition couldn't be harder. `\dimen@` and `\dimen@i` are used as scratch *counters* (to avoid creating a new counter). `\count@`, `\multido@count` and `\multidocount` are also used as scratch counters. `\dimen@ii` is used as a scratch dimension register.

This is probably pretty optimal, given the requirements that leading and trailing spaces be OK in the arguments, and that there need not be any numbers to the left of the decimal.

```
\def\FPadd#1#2#3{%
  \edef\multido@temp{#1..\noexpand\@nil#2}%
  \expandafter\FPadd@\multido@temp..\@nil
  \let#3\multido@temp}
\def\FPsub#1#2{%
  \edef\multido@temp{\noexpand\FPsub@#2\noexpand\@empty}%
  \FPadd{#1}{\multido@temp}}
\def\FPsub@#1{\ifx-#1\else-#1\fi}
\def\FPadd@#1.#2.#3\@nil#4.#5.#6\@nil{%
  \begingroup
    \def\multido@temp{\let\next\relax}%
```

```
\let\next\z@
\afterassignment\multido@temp\count@=#1\next
\dimen@i=0#2sp\relax
\let\next\z@
\afterassignment\multido@temp\multido@count=#4\next
\multidocount=0#5\relax
\dimen@=\number\count@ sp
\count@=\@ne
\dimen@ii=1sp
\FPadd@@@#500000000\@nil
\ifdim#1\dimen@ii<\z@
  \count@=-\tw@
  \dimen@=-\dimen@
\fi
\ifdim#4\dimen@ii<\z@
  \count@=-\count@
  \multido@count=-\multido@count
\fi
\ifnum\count@\z@
  \advance\multido@count\dimen@
  \advance\multidocount\dimen@i
  \ifnum\multidocount<\multido@temp\relax
    \advance\multidocount\multido@temp\relax
  \else
    \advance\multido@count\@ne
  \fi
  \advance\count@-3
\else
  \advance\multido@count-\dimen@
  \advance\multidocount-\dimen@i
  \ifnum\multido@count<\z@
    \multido@count=-\multido@count
    \multidocount=-\multidocount
    \advance\count@\@ne
  \else
    \ifnum\multido@count=\z@
      \ifnum\multidocount<\z@
        \multidocount=-\multidocount
        \advance\count@\@ne
      \fi
    \fi
  \fi
  \ifnum\multidocount<\z@
    \advance\multidocount\multido@temp
    \advance\multido@count-\@ne
  \fi
  \advance\multidocount\multido@temp\relax
```

```
    \fi
    \expandafter\FPadd@@\the\multidocount\@empty\@nil
  \endgroup}
\def\FPadd@@#1#2#3\@nil{\xdef\multido@temp{%
  \ifnum\count@=-1-\fi
  \the\multido@count\ifx#2\@empty\else.#2#3\fi}}
\def\FPadd@@@#1#2#3#4#5#6#7#8#9\@nil{\def\multido@temp{1#9}}

\expandafter\catcode`\@=\theatcode\relax
```

This index is only for the documented code. Underlined numbers refer roughly to the line number of the entry's definition, and all others indicate code lines where it is used.