The `fltpoint` package*

Eckhart Guthöhrlein†

Printed November 12, 2004

Abstract

This package provides commands for simple arithmetic with generic TEX. At the moment, there is support for the basic operations addition, subtraction, multiplication and division as well as for rounding numbers to a given precision.

1 Introduction

The need for calculations inside TEX was encountered when working on some macros to convert positions on a linear scale into angle values, since integer values proved not to be sufficiently exact. Although the capabilities of this package are currently rather limited, they may be of some use if you do not need more than the provided functions. The `rccol` package may serve as an example application; it uses the rounding facilities of this package.

2 User interface

The user commands are divided into two categories: the normal and the register commands. Each command is available in those two variants, as decribed below. At first, we have to agree about the syntax for floating point numbers.

2.1 Syntax of floating point numbers

In the syntax descriptions below, ⟨*fp number*⟩ will be used to denote a number according to the following syntax.

⟨*fp number*⟩ := ⟨*opt signs*⟩⟨*opt digits*⟩⟨*opt dot*⟩⟨*opt digits*⟩

⟨*opt signs*⟩ may be any number of '`+`' and/or '`-`' characters, where each '`-`' toggles the sign of the number. ⟨*opt digits*⟩ may be any number of characters '`0`'...'`9`', and ⟨*opt dot*⟩ is the optional decimal sign. For example, the following inputs for ⟨*fp number*⟩ are valid, resulting into the specified numbers. '`100`' → 100, '`010,98700`' → 10,987, '`-,99`' → −0,99, '`-+-+0001,`' → 1, '' → 0, '`---,50`' → −0,5. As you can see, leading and trailing zeros are removed as far as possible, and an 'empty number' (omitting anything optional) is understood as zero.

*This file has version number v1.1b dated 2004/11/12.

†Send comments or bug-reports to the author via e-mail <e_w_g@web.de>.

There is no syntax checking, so if you do not obey the rules above, you are likely to encounter strange error messages, as well as everything might work properly in some cases. Of course, it is also possible to use a macro as ⟨*fp number*⟩ if it expands to a string satisfying the syntax rules.

2.2 Standard operations

\fpAdd
\fpSub
\fpMul
\fpDiv

The standard commands for binary operations have the following common syntax:

`\fp`⟨*bOp*⟩`{`⟨*command sequence*⟩`}{`⟨*fp number*⟩`}{`⟨*fp number*⟩`}`.

This will perform the operation specified by ⟨*bOp*⟩ with the two given numbers, saving the result in ⟨*command sequence*⟩. Possibilities for ⟨*bOp*⟩ are '`Add`', '`Sub`', '`Mul`' and '`Div`', specifying addition, subtraction, multiplication, and division. Example:

```
\fpAdd{\exmplsum}{100,0}{-99,1}
\fpMul{\exmplprod}{5}{\exmplsum}
```

After this, the results of the computations will be stored in the macros `\exmplsum` and `\exmplprod`, expanding to 0,9 and 4,5.

\fpNeg
\fpAbs

Similar to the binary operations, the unary operations share the common syntax

`\fp`⟨*uOp*⟩`{`⟨*command sequence*⟩`}{`⟨*fp number*⟩`}`.

Possibilities for ⟨*uOp*⟩ are '`Abs`' and '`Neg`', meaning absolute amount and negation.

\fpRound

With `\fpRound{`⟨*command sequence*⟩`}{`⟨*fp number*⟩`}{`⟨*precision*⟩`}`, a number can be rounded to the desired precision (a power of ten). The result is saved in ⟨*command sequence*⟩ as usual.

2.3 Register operations

You may use register variants of all operations, which means that you perform the operation on a register which contains a number. A register is referred to using its name; the name may contain any characters including digits.

\fpRegSet
\fpRegGet

Registers are initialized by assigning them values, using `\fpRegSet`. They can be read out into command sequences using `\fpRegGet`.

```
\fpRegSet{⟨reg name⟩}{⟨fp number⟩}
\fpRegGet{⟨reg name⟩}{⟨command sequence⟩}
```

\fpRegAdd
\fpRegSub
\fpRegMul
\fpRegDiv

The binary operations need two register names. After execution, the first register will hold the result of the specified computation, performed with its former value and the value of the second register.

`\fp`⟨*bOp*⟩`{`⟨*reg name 1*⟩`}{`⟨*reg name 2*⟩`}`

\fpRegAbs
\fpRegNeg

Consequently, the unary operations only need the name of the register.

`\fp`⟨*uOp*⟩`{`⟨*reg name*⟩`}`

\fpRegRound

Rounding of registers is also possible.

`\fpRegRound{`⟨*reg name*⟩`}{`⟨*precision*⟩`}`

\fpRegCopy Furthermore, there is one binary operation only available for registers, this is `\fpRegCopy` which assigns the value of ⟨*reg name 2*⟩ to register ⟨*reg name 1*⟩.

For example, consider the following statements.

```
\fpRegSet{test1}{36}      \fpRegSet{test2}{-3}
\fpRegDiv{test1}{test2}   \fpRegMul{test1}{test1}
\fpRegGet{test1}{\fpresult}
```

After this, `test1` will hold the value 144, which `\fpresult` will expand to.

2.4 Configuration and Parameters

\fpAccuracy The macro `\fpAccuracy` takes one argument (a number), determining the number of digits after the decimal sign, i. e., the accuracy of the computations. The default value is five. At the moment, the name promises too much. The command only affects `\fpDiv` and `\fpRegDiv`.

\fpDecimalSign With `\fpDecimalSign{`⟨*character*⟩`}` you can chose any character for use as the decimal sign. Normally, this will be either a point or a comma; the default is a comma. You can furthermore use the package options `comma` or `point`. The support for options like `english` or `german` has been removed. It will not be added again, and there will be no detection of packages like `babel` or `german`. In my view, a comma is the better choice regardless of the language in question (and it is the ISO standard). On the other hand, many people think that a point should be used even in German texts. So, you have to make an explicit decision.

3 Final Remarks

After the first release, I intended to include the features listed below in the near future. Unfortunately, I didn't have time to do so, and maybe I will never have, since I am currently not interested in extending this package. If I continued the development some day, the first extensions might be what is listed here.

- Extend syntax to support numbers like 1,7E−1 or $2{,}765 \cdot 10^5$ in input and output.
- Formatted, customizable output.
- User access to the comparison of registers.
- A better concept for chosing the accuracy of the computations.
- More operations like e^x, $\sqrt{x}$, $\sin x$, $\ln x \ldots$

Some users have pointed out that the terminus 'floating-point' is not strictly correct for what is provided by the package. Alas! I happily stick to the package name.

If you encounter needs not satisfied by this package, you may wait for the unlikely event of an extension from my part, or you can have a look at the following packages and see if they do what you want:

- `fp` by Michael Mehlich for calculations,
- `numprint` by Harald Harders for formatted printing of numbers.

Finally, the license of this package is LPPL, so feel free to do it yourself.

4 Implementation

4.1 General ideas

The main idea was to represent numbers internally by storing their digits in an array/record-like construction (to be referred to as an array or as a register from now on) whose numbering reflects the decimal position factor of the digit, with some information about the range of the numbering and the sign of the number. An array consists of a couple of command sequences, sharing a common name followed by an element number. E. g., '120.3' means $1\cdot10^2+2\cdot10^1+0\cdot10^0+3\cdot10^{-1}$. So, if the number is to be stored in the array `\exmpl`, the command sequences `\exmpl@2`, `\exmpl@1`, `\exmpl@0` and `\exmpl@-1` will be defined as '`1`', '`2`', '`0`' and '`3`', respectively. The sign information '`+`' will be stored in `\exmpl@sig`. `\exmpl@ul` ('upper limit') will be '`2`', `\exmpl@ll` ('lower limit') will be '`-1`'.

The computations are performed as you do it with paper and pencil. E. g., for an addition, all corresponding digits are summed, taking over anything exceeding ten to the next pair of digits. Thus, there is no limit to the range of numbers or to the number of digits after the decimal sign, except TEX's memory and, probably the limiting factor, your patience.

Initially, the computations were not performed inside of groups, and side-effects were avoided using more counters and constructions like `\xloop` etc. This may make more efficient use of TEX, as far as speed and save stack usage is concerned, but I think that further extensions will be much simpler now without the need to worry about possible side-effects and the surprising result when, once again, something happens you simply did not think of. Furthermore, this provides a simple mechanism of removing temporary stuff from the memory.

But now, let's reveal the code...

4.2 Driver file

The driver file can be generated from `fltpoint.dtx` and then be used to produce the documentation (if you don't like to run LATEX directly over the `dtx`-file).

```
1 ⟨*deccomma⟩
2 \mathchardef\CommaOrdinary="013B
3 \mathchardef\CommaPunct   ="613B
4 \mathcode`,="8000
5 {\catcode`\,=\active
6  \gdef ,{\obeyspaces\futurelet\next\CommaCheck}}
7 \def\CommaCheck{\if\space\next\CommaPunct\else\CommaOrdinary\fi}
8 ⟨/deccomma⟩
9 ⟨*driver⟩
10 \documentclass{ltxdoc}
11 \usepackage{deccomma,fltpoint}
12 %\OnlyDescription
13 \AlsoImplementation
14 \EnableCrossrefs % disable if index is ready
15 \CodelineIndex
16 \RecordChanges
17 %\DisableCrossrefs
18 \newcommand{\fpexample}[1]{%
19    \fpRegSet{fptemp}{#1}%
20    \fpRegGet{fptemp}{\fptemp}%
```

```
    $\mbox{\tt`#1'}\rightarrow\fptemp$}
\begin{document}
    \DocInput{fltpoint.dtx}
\end{document}
⟨/driver⟩
```

4.3 LaTeX package definitions

If used as a LaTeX package, the usual LaTeX preliminaries and some option declarations are necessary.

```
⟨*package⟩
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{fltpoint}[2004/11/12 v1.1b floating point arithmetic]
\DeclareOption{comma}{\AtBeginDocument{\fpDecimalSign,}}
\DeclareOption{point}{\AtBeginDocument{\fpDecimalSign.}}
\ProcessOptions*\relax
\input{fltpoint}
⟨/package⟩
```

4.4 Private letters

`\atcatcode` '`@`' is used for private command sequences. Its catcode is saved in `\atcatcode` to be restored just before `\endinput`.

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

4.5 LaTeX or not?

Check for LaTeX, otherwise provide the `\@ifnextchar` mechanism copied from the LaTeX source, see there for explanation.

```
\ifx\documentclass\relax
\long\def\@ifnextchar#1#2#3{%
  \let\reserved@d=#1%
  \def\reserved@a{#2}%
  \def\reserved@b{#3}%
  \futurelet\@let@token\@ifnch}
\def\@ifnch{%
  \ifx\@let@token\@sptoken
    \let\reserved@c\@xifnch
  \else
    \ifx\@let@token\reserved@d
      \let\reserved@c\reserved@a
    \else
      \let\reserved@c\reserved@b
    \fi
  \fi
  \reserved@c}
\def\:{\let\@sptoken= } \:
\def\:{\@xifnch} \expandafter\def\: {\futurelet\@let@token\@ifnch}
\fi
```

4.6 Additional loop structures

\iloop
\xloop

To be able to nest loop structures without the need for hiding the inner loop(s) in grouped blocks, the constructions `\iloop...\irepeat` and `\xloop...\xrepeat` are defined analogously to Plain TeX's `\loop...\repeat`. `\iloop` will be used 'internally' by macros which are to be used in ordinary `\loop`s or in `\xloop`s. `\xloop` will be used 'externally', surrounding ordinary `\loop`s.

```
\def\iloop#1\irepeat{\def\ibody{#1}\iiterate}
\def\iiterate{\ibody\let\inext=\iiterate\else\let\inext=\relax\fi
   \inext}
\def\xloop#1\xrepeat{\def\xbody{#1}\xiterate}
\def\xiterate{\xbody\let\xnext\xiterate\else\let\xnext\relax\fi\xnext}
```

The following assignments are necessary to make `\loop...\if...\repeat` constructions skippable inside another `\if`.

```
\let\repeat\fi
\let\irepeat\fi
\let\xrepeat\fi
```

4.7 Allocation of registers

\fp@loopcount
\fp@loopcountii
\fp@result
\fp@carryover
\fp@tempcount
\fp@tempcountii

Several count registers are needed. I have tried to keep this number small, which means that, at some points, I may have chosen a less logical or less readable usage of counts. Nevertheless, I do not claim to have minimized the number as far as possible...

`\fp@loopcount` and `\fp@loopcountii` are often, but not always, used for `\loop`s, `\fp@loopcountii` sometimes just stores the finishing number. `\fp@result` and `\fp@carryover` are used to store the intermediate results of computations. `\fp@tempcount` and `\fp@tempcountii` are scratch registers whose values should not be considered to be the same after the use of any macro, except the simple array accession abbreviations starting whith `\ar@`, as explained below.

```
\newcount\fp@loopcount
\newcount\fp@loopcountii
\newcount\fp@result
\newcount\fp@carryover
\newcount\fp@tempcount
\newcount\fp@tempcountii
```

4.8 Communication between macros and groups

\fp@setparam
\fp@param

To pass information from one macro to another, or from inside a group to the outer world, the construction `\fp@setparam{`⟨*information*⟩`}` is used. It saves ⟨*information*⟩ globally in the command sequence `\fp@param`. This mechanism is used, e. g., by `\fp@regcomp`, `\fp@getdigit` to pass their result to the calling macro, or by `\fp@regadd` etc. to make ⟨*information*⟩ survive the end of the current group. Since `\xdef` is used, ⟨*information*⟩ will be fully expanded.

```
\def\fp@setparam#1{\xdef\fp@param{#1}}%
```

4.9 Array accession

\ar@set
\ar@get
\ar@setsig
\ar@getsig
\ar@setul
\ar@getul
\ar@setll
\ar@getll

The idea of arrays using command sequences like `\exmpl@-1` means typing a lot of unreadable `\expandafter`s and `\csname`s, so the following abbreviations were

introduced. They take the base name of the array as the first argument, if needed followed by an element number, for the `set`-commands followed by the third argument to be the (new) value. No checks are performed if the element number is inside the boundaries of the array, nor anything else to ensure the validity of the operation.

`\ar@set` is used to save digits. `\ar@setsig`, `\ar@setul` and `\ar@setll` set sign, upper and lower limit of the array. `\ar@get`, `\ar@getsig`, `\ar@getul` and `\ar@getll` are used to access the respective command sequences.

```
\def\ar@set#1#2#3{\expandafter\edef\csname#1@\number#2\endcsname{%
   \number#3}}
\def\ar@get#1#2{\csname#1@\number#2\endcsname}
\def\ar@setsig#1#2{\expandafter\edef\csname#1@sig\endcsname{#2}}
\def\ar@getsig#1{\csname#1@sig\endcsname}
\def\ar@getul#1{\csname#1@ul\endcsname}
\def\ar@getll#1{\csname#1@ll\endcsname}
\def\ar@setul#1#2{\expandafter\edef\csname#1@ul\endcsname{\number#2}}
\def\ar@setll#1#2{\expandafter\edef\csname#1@ll\endcsname{\number#2}}
```

4.10 Miscellaneous

`\fp@settomax` The macro `\fp@settomax` assigns the maximum of the two numbers given as `#2` and `#3` to the counter `#1`.

```
\def\fp@settomax#1#2#3{%
   \ifnum#2<#3\relax
      #1=#3\relax
   \else
      #1=#2\relax
   \fi
}
```

`\fp@settomin` The macro `\fp@settomin` does the same with the minimum.

```
\def\fp@settomin#1#2#3{%
   \ifnum#2<#3\relax
      #1=#2\relax
   \else
      #1=#3\relax
   \fi
}
```

`\fp@modulo` The macro `\fp@modulo` computes the result of $\#1 \bmod \#2$ and saves it in `\fp@param`.

```
\def\fp@modulo#1#2{%
   \fp@tempcount=#1\relax
   \fp@tempcountii=#1\relax
   \divide\fp@tempcountii#2\relax
   \multiply\fp@tempcountii#2\relax
   \advance\fp@tempcount-\fp@tempcountii
   \edef\fp@param{\number\fp@tempcount}}
```

4.11 Setting and getting register contents

`\fp@regread` `\fp@regread@raw` The macro `\fp@regread` reads the string or command sequence (after expansion) given as `#2` into register `#1`. The main work is done by the subroutine

\fp@readchars, where \fp@tempcount is used to indicate the current position. \fp@arrayname is used to pass #1 to \fp@readchars.

```
\def\fp@regread#1#2{%
   \fp@regread@raw{#1}{#2}%
   \fp@cleanreg{#1}}
\def\fp@regread@raw#1#2{%
```

Initialize \fp@tempcount. Initialize \fp@arrayname. Make #1 positive by default.

```
   \fp@tempcount=0
   \edef\fp@arrayname{#1}%
   \ar@setsig{#1}{+}%
```

Now call \fp@readchars with #2 fully expanded, followed by a decimal sign. The decimal sign is necessary because \fp@readchars expects at least one decimal sign to occur in the given string, so if #2 is, say, 100, this will make it readable. On the other hand, a superficial decimal sign at the end of a number like 1.34 will be ignored.

```
   \edef\fp@scratch{#2\fp@decimalsign}%
   \expandafter\fp@readchars\fp@scratch\end
```

If the first character of #2 has been a decimal sign, the upper limit will be wrong, no pre-point digits will be present. This does not conform the internal syntax and is corrected now.

```
   \ifnum\ar@getul{#1}=-1
      \ar@setul{#1}{0}%
      \ar@set{#1}{0}{0}%
   \fi
```

The n digits before the decimal sign (if any) have been read in from left to right, assigning positions from $0 \ldots n$, so they have to be swapped to their correct positions. This is done with two counters, one starting as 0, the other as n, using \fp@scratch for temporary storage.

```
   \fp@tempcount=0
   \fp@tempcountii=\ar@getul{#1}\relax
   \iloop
   \ifnum\fp@tempcount<\fp@tempcountii
      \edef\fp@scratch{\ar@get{#1}{\fp@tempcountii}}%
      \ar@set{#1}{\fp@tempcountii}{\ar@get{#1}{\fp@tempcount}}%
      \ar@set{#1}{\fp@tempcount}{\fp@scratch}%
      \advance\fp@tempcount by 1
      \advance\fp@tempcountii by -1
   \irepeat
}% end \fp@regread@raw
```

\fp@readchars As mentioned above, this subroutine is called by \fp@regread to do the actual work of reading the given number character after character into the register passed using \fp@arrayname. It will stop if it sees an end token.

```
\def\fp@readchars#1{%
   \ifx#1\end
```

If the condition is true, the token read before has been the final one. So at the end, do not call \fp@readchars any more, and use the current value of \fp@tempcount to assign the correct lower limit to the register.

```
      \let\inext=\relax
```

```
      \ifnum\fp@tempcount<0
         \advance\fp@tempcount by 1
         \ar@setll{\fp@arrayname}{\fp@tempcount}%
      \else
         \ar@setll{\fp@arrayname}{0}%
      \fi
   \else % \ifx#1\end
```

If the condition is false, further characters will follow, so `\fp@readchars` will have to be called again after finishing this character.

```
      \let\inext=\fp@readchars
```

Now check the character and perform the respective actions.

```
      \ifx#1+%
```

An optional '+' has been encountered, nothing to do.

```
      \else
         \ifx#1-%
```

'−' sign, toggle sign.

```
            \if\ar@getsig{\fp@arrayname}-%
               \ar@setsig{\fp@arrayname}{+}%
            \else
               \ar@setsig{\fp@arrayname}{-}%
            \fi
         \else
            \if\noexpand#1\fp@decimalsign%
```

A decimal sign has been encountered. So, if it is the first one, switch to reading afterpoint digits, otherwise ignore it.

```
               \ifnum\fp@tempcount>-1
                  \advance\fp@tempcount by -1
                  \ar@setul{\fp@arrayname}{\fp@tempcount}%
                  \fp@tempcount=-1
               \fi
            \else
```

None of the above characters was encountered, so assume a digit, and read it into the current position. Then step `\fp@tempcount` by $+1$ if prepoint digits are read in, or by -1 if the decimal sign has already been seen.

```
               \ar@set{\fp@arrayname}{\fp@tempcount}{#1}%
               \ifnum\fp@tempcount<0
                  \advance\fp@tempcount by -1
               \else
                  \advance\fp@tempcount by 1
               \fi
            \fi% end \if\noexpand#1\fp@decimalsign
         \fi% end \ifx#1-
      \fi% end \ifx#1+
   \fi% end \ifx#1\end
```

That's all, call `\inext`.

```
   \inext
}% end \fp@readchars
```

`\fp@regget` The macro `\fp@regget` is used to read the contents of the register `#1` into the command sequence `#2`.

```
\def\fp@regget#1#2{%
```

First, we get the sign of the number. If negative, `#2` is initialized as '`-`', otherwise as empty.

```
   \if\ar@getsig{#1}-%
      \def#2{-}%
   \else
      \def#2{}%
   \fi
```

Then we set up `\fp@tempcount` as the counter for an `\iloop`, starting at the upper limit of `#1`.

```
   \fp@tempcount=\ar@getul{#1}\relax
   \iloop
```

If the `\fp@tempcount` is -1, we have to append a decimal sign.

```
      \ifnum\fp@tempcount=-1
         \edef#2{#2\fp@decimalsign}%
      \fi
```

Now append the corresponding digit.

```
      \edef#2{#2\ar@get{#1}{\fp@tempcount}}%
```

And repeat if the lower limit of `#1` is not yet reached.

```
   \ifnum\fp@tempcount>\ar@getll{#1}\relax
      \advance\fp@tempcount by -1
   \irepeat
}% end \def\fp@regget
```

`\fp@cleanreg` The macro `\fp@cleanreg` will clean up the given register. This means that leading and trailing zeros will be removed, and that -0 will be turned into $+0$ to be recognised as equal later on.

```
\def\fp@cleanreg#1{%
```

First, we will iterate until all leading zeros have been removed, except for digit 0 that it is expected to be '`0`' for all numbers n with $-1 < n < 1$.

```
   \fp@tempcount=\ar@getul{#1}\relax
   \iloop
   \ifnum\fp@tempcount>0
      \ifnum\ar@get{#1}{\fp@tempcount}=0
```

If this is true, the first digit is a zero and is 'removed' by changing the upper limit. It is not necessary to erase it by setting the array element to `\empty` or something like that, because it will not be looked at any more.

```
         \advance\fp@tempcount by -1
         \ar@setul{#1}{\fp@tempcount}%
      \else
```

So the condition is false, the first digit is not a zero and the following ones need not to be looked at.

```
         \fp@tempcount=0
      \fi
   \irepeat
```

Similarly, the trailing zeros are removed.

```
   \fp@tempcount=\ar@getll{#1}\relax
   \iloop
```

```
    \ifnum\fp@tempcount<0
       \ifnum\ar@get{#1}{\fp@tempcount}=0
          \advance\fp@tempcount by 1
          \ar@setll{#1}{\fp@tempcount}%
       \else
          \fp@tempcount=0
       \fi
    \irepeat
```

Now check if the number is zero, using $(\text{x@ll} = \text{x@ul}) \wedge (\text{x@0} = 0) \iff \text{x} = 0$, and set the sign to '+' if this is the case.

```
    \ifnum\ar@getll{#1}=\ar@getul{#1}\relax
       \ifnum\ar@get{#1}{0}=0\relax
          \ar@setsig{#1}{+}%
       \fi
    \fi
}% end \fp@regclean
```

`\fp@getdigit` The macro `\fp@getdigit` will return the digit number `#2` of register `#1` using `\fp@setparam`. If `#2` is outside the boundaries of the array, '0' is returned. (Which is not only sensible, but also mathematically correct.)

```
\def\fp@getdigit#1#2{%
    \ifnum#2<\ar@getll{#1}\relax
       \fp@setparam0%
    \else
       \ifnum#2>\ar@getul{#1}\relax
          \fp@setparam0%
       \else
          \fp@setparam{\ar@get{#1}{#2}}%
       \fi
    \fi
}% end \fp@getdigit
```

`\fp@shiftright` The macro `\fp@shiftright` takes register `#1` and shifts the decimal sign `#2` digits to the right (`#2` may be negative or zero, too, so there is no need for a `\fp@shiftleft`). The digits are read into `\fp@shiftnum`, inserting the decimal sign at the new place. Then, `\fp@shiftnum` is read into `#1` via `\fp@regread`.

```
\def\fp@shiftright#1#2{%
```

First, save the value of `#2` in `\fp@shiftamount`. This makes it possible to say, e. g., `\fpshiftright{exmpl}{\fp@tempcount}` without side-effects.

```
    \edef\fp@shiftamount{\number#2}%
```

Now, determine the start position. The maximum of the upper limit and `-\fp@shiftamount` is used in order to allow the decimal sign of, e. g., 1.1 to be shifted -5 digits to the right.

```
    \fp@settomax{\fp@tempcount}{\ar@getul{#1}}{-\fp@shiftamount}%
```

Similarly, determine the stop position.

```
    \fp@settomin{\fp@tempcountii}{\ar@getll{#1}}{-\fp@shiftamount}%
```

Now, initialize `\fp@shiftnum` and begin the `\iloop`. Read digit after digit using `\fp@getdigit`, therefore getting a '0' outside the boundaries. Insert the decimal sign at the new position given by `-\fp@shiftamount`.

```
    \def\fp@shiftnum{}%
```

```
223    \iloop
224       \fp@getdigit{#1}{\fp@tempcount}%
225       \edef\fp@shiftnum{\fp@shiftnum\fp@param}%
226       \ifnum\fp@tempcount=-\fp@shiftamount\relax
227          \edef\fp@shiftnum{\fp@shiftnum\fp@decimalsign}%
228       \fi
229    \ifnum\fp@tempcount>\fp@tempcountii
230       \advance\fp@tempcount by -1
231    \irepeat
```

Finally, assign the value to `#1`.

```
232    \fp@regread{#1}{\fp@shiftnum}%
233 }% end \fp@shiftright
```

`\fp@firstnonzero` The macro `\fp@firstnonzero` returns the first non-zero digit of register `#1` via `\fp@setparam`.

```
234 \def\fp@firstnonzero#1{%
```

If `#1` is zero, the `\iloop` below will run infinitely, so this case has to be checked separately by comparing `#1` to the internal register `@0` which holds zero. '0' is returned if `#1` is zero.

```
235    \fp@regcomp{#1}{@0}%
236    \if\fp@param=%
237       \fp@setparam0%
```

Otherwise, each digit is checked, starting at the upper limit, and the position of first digit differing from zero is returned in `\fp@param`.

```
238    \else
239       \fp@tempcount=\ar@getul{#1}\relax%
240       \fp@tempcountii=\ar@getll{#1}\relax%
241       \iloop
242          \ifnum\ar@get{#1}{\fp@tempcount}>0
243             \fp@setparam{\number\fp@tempcount}%
244             \fp@tempcount=\fp@tempcountii
245          \fi
246       \ifnum\fp@tempcount>\fp@tempcountii
247          \advance\fp@tempcount by -1
248       \irepeat
249    \fi
250 }% end \fp@firstnonzero
```

4.12 Comparison of registers

`\fp@regcomp` The macro `\fp@regcomp` compares the two specified registers. It saves the result of the comparison (either '<', '>', or '=') in `\fp@param`. First, it checks whether the two numbers have the same sign or not. If not, the comparison is very easy, otherwise `\fp@regcomp@main` is called to do the work.

```
251 \def\fp@regcomp#1#2{%
252    {%
253       \if\ar@getsig{#1}-%
254          \if\ar@getsig{#2}-%
255             \fp@regcomp@main{#1}{#2}<>%
256          \else
257             \fp@setparam{<}%
```

```
         \fi
      \else
         \if\ar@getsig{#2}-%
            \fp@setparam{>}%
         \else
            \fp@regcomp@main{#1}{#2}><%
         \fi
      \fi
   }%
}
```

`\fp@regcomp@main` The macro `\fp@regcomp@main` takes four parameters: The two registers to be compared, and two tokens to be used as result. This is needed because if, e. g., two numbers have the same sign and are equal for all positions greater than 10^2, and number 1 has '9' at position 10^2 and number 2 has '5', then the result must be '<' if $n_1 < n_2 < 0$, but '>' if $n_1 > n_2 > 0$.

First, the range of digits to compare is determined. Then, each pair of digits is compared. If different, `\fp@param` is set and the loop is terminated by setting the loop counter to the stop position. If the digits are equal and there are no more digits to compare, the numbers are equal.

```
\def\fp@regcomp@main#1#2#3#4{%
   \fp@settomax{\fp@loopcount}{\ar@getul{#1}}{\ar@getul{#2}}%
   \fp@settomin{\fp@loopcountii}{\ar@getll{#1}}{\ar@getll{#2}}%
   \loop
      \fp@getdigit{#1}{\fp@loopcount}%
      \fp@tempcount=\fp@param\relax
      \fp@getdigit{#2}{\fp@loopcount}%
      \fp@tempcountii=\fp@param\relax
      \ifnum\fp@tempcount<\fp@tempcountii
         \fp@setparam{#4}%
         \fp@loopcount=\fp@loopcountii
      \else
         \ifnum\fp@tempcount>\fp@tempcountii
            \fp@setparam{#3}%
            \fp@loopcount=\fp@loopcountii
         \else
            \ifnum\fp@loopcount=\fp@loopcountii
               \fp@setparam{=}%
            \fi
         \fi
      \fi
   \ifnum\fp@loopcount>\fp@loopcountii
      \advance\fp@loopcount by -1
   \repeat
}% end \fp@regcomp@main
```

4.13 Unary Operations

`\fp@regabs` The macro `\fp@regabs` turns register `#1` into its amount. This is rather trivial: just set the sign to '+'.

```
\def\fp@regabs#1{%
   \ar@setsig{#1}{+}%
}
```

\fp@regneg The macro \fp@regneg negates register #1. It checks whether the actual sign is '+' or '-' and sets it to its opposite, except that nothing is done if the number is zero.

```
\def\fp@regneg#1{%
   \if\ar@getsig{#1}-%
      \ar@setsig{#1}{+}%
   \else
      \fp@regcomp{#1}{@0}%
      \if\fp@param=%
      \else
         \ar@setsig{#1}{-}%
      \fi
   \fi
}
```

\fp@reground The macro \fp@reground rounds register #1 with a target accuracy given as #2 (as a power of ten).

```
\def\fp@reground#1#2{%
```

Fist, if the desired accuracy is smaller than the lower limit of #1, nothing has to be done.

```
   \ifnum#2>\ar@getll{#1}\relax
      {%
```

Otherwise, we check the following digit. If it is greater than four, we have to advance digit #2 before truncating the number. This means adding $10^{\#2}$ for positive #1 and subtracting $10^{\#2}$ for negative #1.

```
         \fp@tempcount=#2\relax
         \advance\fp@tempcount by -1
         \fp@getdigit{#1}{\fp@tempcount}%
         \ifnum\fp@param>4
            \fp@regcopy{fp@temp}{@1}%
            \fp@shiftright{fp@temp}{#2}%
            \fp@regcomp{#1}{@0}%
            \if\fp@param<%
               \fp@regneg{fp@temp}%
            \fi
            \fp@regadd{#1}{fp@temp}%
         \fi
```

Afterwards, we set the lower limit to #2. If #2 is greater than zero, we set the lower limit and all digits n with $0 \leq n <$ #2 to zero. Then we read the number using \fp@regget, make it globally available and read it into #1 after finishing the local group.

```
         \ifnum#2>0
            \fp@loopcount=#2\relax
            \iloop
               \ifnum\fp@loopcount>0
                  \advance\fp@loopcount by -1
                  \ar@set{#1}{\fp@loopcount}{0}%
            \irepeat
            \ar@setll{#1}{0}%
         \else
            \ar@setll{#1}{#2}%
```

```
        \fi
        \fp@regget{#1}{\fp@scratch}%
        \fp@setparam\fp@scratch
      }%
      \fp@regread{#1}{\fp@param}%
  \fi
} % end \fp@reground
```

4.14 Binary operations

`\fp@regcopy` The macro `\fp@regcopy` assigns the value of register `#2` to register `#1`. This is done simply by reading register `#2` into a scratch control sequence and then reading this into register `#1`.

```
\def\fp@regcopy#1#2{%
  \fp@regget{#2}{\fp@scratch}%
  \fp@regread{#1}{\fp@scratch}%
}
```

`\fp@regadd` The macro `\fp@regadd` adds the value of register `#2` to register `#1`.

```
\def\fp@regadd#1#2{%
  {%
```

First, check whether the two numbers have the same sign.

```
    \if\ar@getsig{#1}\ar@getsig{#2}%
```

If the two numbers have the same sign, the addition can be done by adding each two corresponding digits and a possible carryover, starting at min(ll1,ll2), ending at max(ul1,ul2). Those values are saved in `\fp@add@start` and `\fp@add@finish`.

```
      \fp@settomin{\fp@loopcount}{\ar@getll{#1}}{\ar@getll{#2}}%
      \edef\fp@add@start{\number\fp@loopcount}%
      \fp@settomax{\fp@tempcount}{\ar@getul{#1}}{\ar@getul{#2}}%
      \edef\fp@add@finish{\number\fp@tempcount}%
```

Initialize `\fp@carryover`.

```
      \fp@carryover=0
```

Now start the main loop. Each digit is computed in counter `\fp@result` as the sum of the corresponding digits plus the carryover from the previous pair. If the sum is greater than 10, it is reduced by 10 and `\fp@carryover` is set to 1. (No sum greater than 19 is possible.)

```
      \loop
        \fp@getdigit{#1}{\fp@loopcount}%
        \fp@result=\fp@param\relax
        \fp@getdigit{#2}{\fp@loopcount}%
        \advance\fp@result by \fp@param\relax
        \advance\fp@result by \fp@carryover
        \ifnum\fp@result>9
          \fp@carryover=1
          \advance\fp@result by -10
        \else
          \fp@carryover=0
        \fi
        \ar@set{#1}{\fp@loopcount}{\fp@result}%
      \ifnum\fp@loopcount<\fp@add@finish\relax
```

```
            \advance\fp@loopcount by 1
         \repeat
```

If the last pair had a carryover, take it into account. Then adjust the lower and upper limit of the result.

```
         \ifnum\fp@carryover>0
            \advance\fp@loopcount by 1
            \ar@set{#1}{\fp@loopcount}{\fp@carryover}%
         \fi
         \ar@setll{#1}{\fp@add@start}%
         \ar@setul{#1}{\fp@loopcount}%
```

Finally, save the result in `\fp@param` to make it survive the endgroup character after `\fi`.

```
         \fp@regget{#1}{\fp@scratch}%
         \fp@setparam\fp@scratch
```

That's it. But if the two numbers have different signs, the situation is a bit more complicated. In this case, the amounts of `#1` and `#2` are saved in two temporary registers (`fp@tempi` and `fp@tempii`). The smaller one is subtracted from the larger one, and the sign of the result is adjusted according to the sign of `#1` and `#2`. This is done by the subroutine `\fp@regadd@sub`, which also takes care of saving the result in `\fp@param`.

```
      \else % \if sign
         \fp@regcopy{fp@tempi}{#1}%
         \fp@regcopy{fp@tempii}{#2}%
         \fp@regabs{fp@tempi}%
         \fp@regabs{fp@tempii}%
         \fp@regcomp{fp@tempi}{fp@tempii}%
         \if\fp@param>%
            \fp@regadd@sub{#1}{fp@tempi}{fp@tempii}%
         \else
            \fp@regadd@sub{#2}{fp@tempii}{fp@tempi}%
         \fi
      \fi % end \if sign
```

Now end the group to keep everything local, and read the result in `\fp@param` into register `#1`.

```
   }%
   \fp@regread{#1}{\fp@param}%
}% end \fp@regadd
```

`\fp@regadd@sub` The macro `\fp@regadd@sub` is a subroutine of `\fp@regadd`.

```
\def\fp@regadd@sub#1#2#3{%
```

First, subtract `#3` from `#2`. The restriction `#2` > `#3` is ensured by the calling `\fp@regadd`.

```
   \fp@regsub@restricted{#2}{#3}%
```

`#1` is the original number of which `#2` is the amount. So, if it is negative, the final result also has to be negative. This is done by the following four lines.

```
   \fp@regcomp{#1}{@0}%
   \if\fp@param<%
      \fp@regneg{#2}%
   \fi
```

Now, the final result is stored in #2. Make it globally available using \fp@setparam.

```
    \fp@regget{#2}{\fp@scratch}%
    \fp@setparam\fp@scratch
}% end \fp@regadd@sub
```

\fp@regsub@restricted The macro \fp@regsub@restricted does the actual work of subtracting #2 from #1, provided that #1 is greater than #2. It is called by \fp@regadd@sub and by \fp@regdiv.

```
\def\fp@regsub@restricted#1#2{%
```

First, we start a group to keep counters etc. local. Then, we determine the start and end position for the loop, as above for \fp@regadd.

```
    {%
       \fp@settomin{\fp@loopcount}{\ar@getll{#1}}{\ar@getll{#2}}%
       \edef\fp@lowermin{\number\fp@loopcount}%
       \fp@settomax{\fp@tempcount}{\ar@getul{#1}}{\ar@getul{#2}}%
       \edef\fp@uppermin{\number\fp@tempcount}%
```

Now subtract the corresponding digits, taking into account a possible carryover.

```
       \fp@carryover=0
       \loop
          \fp@getdigit{#1}{\fp@loopcount}%
          \fp@result=\fp@param\relax
          \fp@getdigit{#2}{\fp@loopcount}%
          \advance\fp@result by -\fp@param\relax
          \advance\fp@result by \fp@carryover
```

If the result is < 0, add 10 to the result and set the carryover to -1.

```
          \ifnum\fp@result<0
             \fp@carryover=-1
             \advance\fp@result by 10
          \else
             \fp@carryover=0
          \fi
```

Now save the result and repeat if there are further digits.

```
          \ar@set{#1}{\fp@loopcount}{\fp@result}%
       \ifnum\fp@loopcount<\fp@uppermin\relax
          \advance\fp@loopcount by 1
       \repeat
```

If there is a carryover for the last two digits, take it into account.

```
       \ifnum\fp@carryover=-1
          \advance\fp@loopcount by 1
          \ar@set{#1}{\fp@loopcount}{-1}%
       \fi
```

Now adjust the upper and lower limit of the result, and save it in \fp@param.

```
       \ar@setll{#1}{\fp@lowermin}%
       \ar@setul{#1}{\fp@loopcount}%
    \fp@regget{#1}{\fp@scratch}%
    \fp@setparam\fp@scratch
    }%
```

Finally, assign the result to #1 inside the current group.

```
    \fp@regread{#1}{\fp@param}%
}% end \fp@regsub@restricted
```

\fp@regsub The macro \fp@regsub subtracts register #2 from register #1. This is done by negating #2 inside a group and calling \fp@regadd.

```
\def\fp@regsub#1#2{%
   {%
      \fp@regneg{#2}%
      \fp@regadd{#1}{#2}%
      \fp@regget{#1}{\fp@scratch}%
      \fp@setparam\fp@scratch
   }%
   \fp@regread{#1}{\fp@param}%
}
```

\fp@regmul The macro \fp@regmul multiplies the value of register #1 with the value of register #2.

```
\def\fp@regmul#1#2{%
   {%
```

First, we initialize the temporary register fp@temp1 as zero; it will be used to hold the results so far. Then we start the outer \xloop which will run through all digits of #2, beginning at the lower limit.

```
      \fp@regcopy{fp@temp1}{@0}%
      \fp@loopcountii=\ar@getll{#2}\relax
      \xloop
```

Then we initialize the inner loop, which multplies the current digit of #2 with #1 digit after digit, saving the result in \fp@newnum.

```
         \fp@loopcount=\ar@getll{#1}\relax
         \fp@carryover=0
         \def\fp@newnum{}%
         \loop
            \fp@result=\ar@get{#2}{\fp@loopcountii}\relax
            \multiply\fp@result by \ar@get{#1}{\fp@loopcount}\relax
            \advance\fp@result by \fp@carryover
```

If the result is greater than 9, we set the carryover as (\fp@result mod 10) and the result to (\fp@result div 10).

```
            \ifnum\fp@result>9
               \fp@carryover=\fp@result
               \divide\fp@carryover by 10
               \fp@tempcount=\fp@carryover
               \multiply\fp@tempcount by 10
               \advance\fp@result by -\fp@tempcount
            \else
               \fp@carryover=0
            \fi
            \edef\fp@newnum{\number\fp@result\fp@newnum}%
         \ifnum\fp@loopcount<\ar@getul{#1}\relax
            \advance\fp@loopcount by 1
         \repeat
         \edef\fp@newnum{\number\fp@carryover\fp@newnum}%
         \fp@regread{fp@temp2}{\fp@newnum}%
```

Now fp@temp2 holds the partial result for this digit of #2. We have to multiply it with 10^n, if n is the number of digits of #2 completed so far. This is done by calling \fp@shiftright with $-n$ as second argument.

```
        \fp@tempcount=\fp@loopcountii
        \advance\fp@tempcount by -\number\ar@getll{#2}\relax
        \fp@shiftright{fp@temp2}{\fp@tempcount}%
```

Now we add `fp@temp2` to the results so far and iterate if there are further digits.

```
        \fp@regadd{fp@temp1}{fp@temp2}%
    \ifnum\fp@loopcountii<\ar@getul{#2}\relax
        \advance\fp@loopcountii by 1
    \xrepeat
```

The final result of the multiplication will have as much afterpoint digits as `#1` and `#2` have together. Adjust this.

```
    \fp@tempcount=\ar@getll{#1}\relax
    \advance\fp@tempcount by \ar@getll{#2}\relax
    \fp@shiftright{fp@temp1}{\fp@tempcount}%
```

If `#1` and `#2` have different signs, the result is negative, otherwise positive.

```
    \if\ar@getsig{#1}\ar@getsig{#2}%
    \else
        \fp@regneg{fp@temp1}%
    \fi
```

Finally, save the result via `\fp@setparam` and assign it to `#1` after the end of the group.

```
  \fp@regget{fp@temp1}{\fp@scratch}%
  \fp@setparam\fp@scratch
  }%
  \fp@regread{#1}{\fp@param}%
} % end \fp@regmul
```

`\fp@regdiv` The macro `\fp@regdiv` divides register `#1` by register `#2`. It works by repeated subtraction.

```
\def\fp@regdiv#1#2{%
  {%
```

The amount of the two numbers is read into the two temporary registers `fp@temp1` and `fp@temp2`.

```
    \fp@regcopy{fp@temp1}{#1}%
    \fp@regcopy{fp@temp2}{#2}%
    \fp@regabs{fp@temp1}%
    \fp@regabs{fp@temp2}%
```

First, we determine the initial shift for `fp@temp2`. This is the shift which will make `fp@temp2` have as many digits before the decimal sign as `fp@temp1`. `\fp@firstnonzero` is used, because the upper limit need not be the first non-zero digit.

```
    \fp@firstnonzero{fp@temp1}%
    \fp@loopcountii=\fp@param\relax
    \fp@firstnonzero{fp@temp2}%
    \advance\fp@loopcountii by -\fp@param\relax
    \fp@shiftright{fp@temp2}{\fp@loopcountii}%
```

Now we initialize `\fp@divnum` which will hold the result. If `\fp@loopcountii` is smaller than zero, i. e., if the first digit of the result that will be computed is after the decimal sign, we have to initialize `\fp@divnum` with the decimal sign as well as with an appropriate number of zeros following it.

```
      \def\fp@divnum{}%
      \ifnum\fp@loopcountii<0
         \fp@tempcount=\fp@loopcountii
         \loop
         \ifnum\fp@tempcount<-1
            \edef\fp@divnum{0\fp@divnum}%
            \advance\fp@tempcount by 1
         \repeat
         \edef\fp@divnum{\fp@decimalsign\fp@divnum}%
      \fi
```

The main loop follows. Each digit is determined by subtracting the divisor n times from the dividend until the result is smaller than the divisor. This is done only if `\fp@loopcountii` plus one is greater than `-\fp@accuracy`. If the divisor is equal to the dividend, the division is complete and the `\xloop` is terminated. Therefore, `\fp@accuracy` is locally set to '0', so that possibly following zeros are computed until the digit representing 10^0. At the end, the divisor is divided by 10, and the next digit follows.

```
      \xloop
      \fp@tempcount=\fp@loopcountii
      \advance\fp@tempcount by 1
      \ifnum\fp@tempcount>-\fp@accuracy\relax
         \fp@loopcount=0
         \loop
            \fp@regcomp{fp@temp2}{fp@temp1}%
            \if\fp@param=%
               \def\fp@accuracy{0}%
               \gdef\fp@param{<}%
            \fi
         \if\fp@param<%
            \fp@regsub@restricted{fp@temp1}{fp@temp2}%
            \advance\fp@loopcount by 1
         \repeat
         \ifnum\fp@loopcountii=-1
            \edef\fp@divnum{\fp@divnum\fp@decimalsign}%
         \fi
         \edef\fp@divnum{\fp@divnum\number\fp@loopcount}%
         \fp@shiftright{fp@temp2}{-1}%
         \advance\fp@loopcountii by -1
      \xrepeat
```

The sign of the result is set according to the signs of `#1` and `#2`.

```
      \if\ar@getsig{#1}\ar@getsig{#2}%
         \fp@regread{fp@temp1}{\fp@divnum}%
      \else
         \fp@regread{fp@temp1}{-\fp@divnum}%
      \fi
```

Now save the result in `\fp@param`. After endgroup, read it into `#1`.

```
      \fp@regget{fp@temp1}{\fp@scratch}%
      \fp@setparam\fp@scratch
   }%
   \fp@regread{#1}{\fp@param}%
}
```

4.15 User interface

`\fp@call@bin` The macro `\fp@call@bin` is a common calling command used by the user commands for binary operations. It reads the values given in `#2` and `#3` into temporary registers, performs the operation specified in `#4`, and finally assigns the result to the command sequence given as `#1`.

```
\def\fp@call@bin#1#2#3#4{%
   {%
      \fp@regread{fp@user1}{#2}%
      \fp@regread{fp@user2}{#3}%
      \csname fp@reg#4\endcsname{fp@user1}{fp@user2}%
      \fp@regget{fp@user1}{\fp@scratch}%
      \fp@setparam\fp@scratch
   }%
   \edef#1{\fp@param}%
}
```

`\fpAdd` As described above, the main work is done by `\fp@call@bin`, so this macro reduces to passing the parameters and specifying the desired operation.

```
\def\fpAdd#1#2#3{\fp@call@bin{#1}{#2}{#3}{add}}
```

`\fpSub` Just like `\fpAdd`.

```
\def\fpSub#1#2#3{\fp@call@bin{#1}{#2}{#3}{sub}}
```

`\fpMul` Just like `\fpAdd`.

```
\def\fpMul#1#2#3{\fp@call@bin{#1}{#2}{#3}{mul}}
```

`\fpDiv` Just like `\fpAdd`.

```
\def\fpDiv#1#2#3{\fp@call@bin{#1}{#2}{#3}{div}}
```

`\fp@call@un` Similarly, the unary operations `\fpAbs` and `\fpNeg` refer to the common macro `\fp@call@un`.

```
\def\fp@call@un#1#2#3{%
   {%
      \fp@regread{fp@user1}{#2}%
      \csname fp@reg#3\endcsname{fp@user1}%
      \fp@regget{fp@user1}{\fp@scratch}%
      \fp@setparam\fp@scratch
   }%
   \edef#1{\fp@param}%
}
```

`\fpAbs` Pass the information and specify the action.

```
\def\fpAbs#1#2{\fp@call@un{#1}{#2}{abs}}
```

`\fpNeg` Just like `\fpAbs`.

```
\def\fpNeg#1#2{\fp@call@un{#1}{#2}{neg}}
```

`\fpRound` This macro does not fit into the scheme, so it has to be defined seperately.

```
\def\fpRound#1#2#3{%
   {%
      \fpRegSet{fp@user1}{#2}%
      \fpRegRound{fp@user1}{#3}%
```

```
      \fpRegGet{fp@user1}{\fp@scratch}%
      \fp@setparam\fp@scratch
    }%
    \edef#1{\fp@param}%
}
```

`\fpRegSet` The register operations `\fpRegSet`, `\fpRegGet`, `\fpRegAdd`, `\fpRegSub`, `\fpRegMul`, `\fpRegDiv`, `\fpRegAbs`, `\fpRegNeg`, `\fpRegCopy` and `\fpRegRound` have the same syntax as the internal variants, so their definitions reduce to passing the parameters. The register name is always given as the first parameter.

```
\def\fpRegSet#1#2{\fp@regread{#1}{#2}}
```

`\fpRegGet` As described above.

```
\def\fpRegGet#1#2{\fp@regget{#1}{#2}}
```

`\fpRegAdd` As described above.

```
\def\fpRegAdd#1#2{\fp@regadd{#1}{#2}}
```

`\fpRegSub` As described above.

```
\def\fpRegSub#1#2{\fp@regsub{#1}{#2}}
```

`\fpRegMul` As described above.

```
\def\fpRegMul#1#2{\fp@regmul{#1}{#2}}
```

`\fpRegDiv` As described above.

```
\def\fpRegDiv#1#2{\fp@regdiv{#1}{#2}}
```

`\fpRegAbs` As described above.

```
\def\fpRegAbs#1{\fp@regabs{#1}}
```

`\fpRegNeg` As described above.

```
\def\fpRegNeg#1{\fp@regneg{#1}}
```

`\fpRegCopy` As described above.

```
\def\fpRegCopy#1#2{\fp@regcopy{#1}{#2}}
```

`\fpRegRound` As described above.

```
\def\fpRegRound#1#2{\fp@reground{#1}{#2}}
```

`\fpAccuracy` `\fp@accuracy` The user command `\fpAccuracy` `\edef`s the internal parameter `\fp@accuracy`, which stores the maximum number of digits after the decimal sign, i. e., the minimum for the lower limit of fp numbers. At the moment, `\fp@accuracy` does not affect the accuracy of any operation except `\fp@regdiv`. In fact, it was introduced when the definition of a termination condition for the loop was not possible without an externally given limit. `\fp@accuracy` is initialized to '5' digits after the decimal sign.

```
\def\fpAccuracy#1{\edef\fp@accuracy{#1}}
\fpAccuracy{5}
```

\fpDecimalSign
\fp@decimalsign

The command \fpDecimalSign allows the user to select any character for use as the decimal sign. The character is stored in \fp@decimalsign. Normally, the decimal sign will be either '.' or ','; a comma is the default. (Take a look at ISO 31-0, part 3.3.2, if you dislike this.)

```
\def\fpDecimalSign#1{\edef\fp@decimalsign{#1}}
\fpDecimalSign{,}
```

\fpThousandsep
\fp@thousandsep

Those macros are used to define and store a thousand seperator used by \fp@regoutput. By default, there is none.

```
\def\fpThousandSep#1{\edef\fp@thousandsep{#1}}
\fpThousandSep{}
```

4.16 Constants

@0
@1

The number zero ist stored in register @0, the number one in register @1.

```
\fp@regread{@0}{0}
\fp@regread{@1}{1}
```

4.17 Finish

Finally, restore the catcode of '@' and \endinput.

```
\catcode`\@=\atcatcode\relax
\endinput
⟨/fltmain⟩
```

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.

Change History