

quire.tex

Macros for making booklets, printing double pages, and printing outlines and crop marks.

Version: 1.0 (19 January 1990) (doc: 10 Sep 1991)

Copyright © 1991 Marcel R. van der Goot

A quire is a “set of folded sheets (as of a book) fitting one within another” [Webster’s]. Such a set of sheets forms a little booklet. Larger books usually consist of a number of quires, with 20 to 40 pages each. The macros in *quire.tex* help you to turn your own texts into such booklets. (The macros can also be used for some other things, such as printing crop marks.) *Quire.tex* is designed to minimize the interference with other macro packages, so that, for instance, it can be used with plain $\text{T}_{\text{E}}\text{X}$ as well as with $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$. You do not need any special programs or special equipment to print quires, but it is convenient if you can print pages in “landscape mode.” Whether you can print in landscape mode depends on your device driver and printer (most printers support this), it is outside the control of $\text{T}_{\text{E}}\text{X}$. The reason for using landscape mode is that, because sheets are folded, two pages have to be printed next to each other on the same sheet. In “portrait mode” this allows only very narrow pages. Making quires is easier if you have a large version of $\text{T}_{\text{E}}\text{X}$ (i.e., one with lots of memory).

Before we explain what the macros do, let us emphasize what they do not do. *Quire.tex* does *not* change the layout of your pages, look at page numbers, print odd and even pages differently, or print small pages. These are all things that you have to take care of with other macros or style files. What *quire.tex* does is to take your pages, reorder them, print two per sheet, and draw outlines around them. As an example, assume we want to make a booklet consisting of five sheets folded together. With two pages next to each other, and with double sided printing, this gives us a booklet consisting of 20 pages. The first page of the five folded sheets is page 1. When we open the booklet we get page 2, printed on the back of the same sheet as page 1. Page 2 occupies the left-hand side of that sheet. The page on the right-hand side is not page 3, which is printed on the next sheet, but page 19. This reordering of pages is the main task of *quire.tex*. For this example, the following figure depicts two of the sheets generated by *quire.tex* (these are the last two sheets that are generated).



Once again, *quire.tex* does not look at the page numbers you print. What we call here “page 1” is simply what the first page of output would be if you were not using *quire.tex*. To make a booklet, the two double pages in the figure should be printed on the opposite sides of the same sheet of paper. There are two ways of doing that. One is to print all the sheets and then use a photocopier to make double sided prints. The second is to print all even sheets, walk to your printer and reinsert the stack of printed sheets upside down, then print all the odd sheets. Many device drivers can be instructed to print only even or odd sheets. But in case your installation does not support this, you can say `\shonly1` at the beginning of your file to instruct *quire.tex* to only generate odd sheets; `\shonly2` generates only even sheets; and `\shonly0`, which is the default, generates both.

Before you can use any of the macros, you have to `\input quire*`. Then you give values to some of the parameters (which are discussed later), and you can call, for instance, `\quire`. All of this should be done preceding the actual text of your file; in $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ all of this should be put in the preamble. If you are

* Although *quire.tex* can be used with $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$, we have made no attempts to $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ ify the macros. So yes, in $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ you use the macros exactly as described here.

using \LaTeX , you must say $\text{\backslash latexquire}$ somewhere in the preamble, regardless of what other macros you are using. The macro \backslash quire is used to start typesetting of a quire. It takes one argument, the number of pages in the quire. Hence you have to \TeX your document at least once before you use \backslash quire , to determine the number of pages. That also helps to separate errors in your input from problems caused by *quire.tex*. Say your document consists of 19 pages, numbered 1...19. When you say

```
 $\backslash quire\{19\}$ 
```

\TeX will produce 10 sheets. The first one is sheet 10 with pages 10 and 11, the second is sheet 9 with pages 12 and 9, etc. The last sheet has pages 20 and 1, where page 20 is blank.

This example illustrates that if there are not enough pages for a complete quire, \TeX will insert empty pages at the end of your text. If, on the other hand, there are more pages than fit in a quire, \TeX will just generate more than one quire. For instance, a 250-page book can be printed in 13 quires with $\text{\backslash quire}\{20\}$. The mechanism through which \TeX supplies extra blank pages at the end of your text is called \TeX 's "endgame." These extra pages are passed to \TeX 's output routine, which usually means that they will have headlines and footlines. You can bypass the output routine by saying $\text{\backslash endquire}$ at the end of your text, in which case the blank pages will really be blank pages (you can say $\text{\backslash endquire}$ even if you are not using \backslash quire). In \LaTeX you should *not* use $\text{\backslash endquire}$, it is implied by the $\text{\backslash end}\{\text{document}\}$ that terminates your file. Sometimes you may want to start your text with an empty page. Here are two simple ways of generating an empty starting page (in plain \TeX).

```
 $\backslash line\{\}\backslash vfill\backslash eject\ \% \text{ or:}$   
 $\backslash shipout\ \backslash vbox\{\}$ 
```

The method with $\text{\backslash shipout}$ bypasses the output routine.

Let us turn to the parameters that determine the output of *quire.tex*. You have to take care that you indeed generate small pages. The page size is normally determined by \backslash hsize and \backslash vsize , but depending on the macro package you are using, other parameters may also be important. However, *quire.tex* does not look at the values of these parameters, they only serve to determine the size of the text area (i.e., the printed part of the page). In the following, "page" refers to the small pages you print, "sheet" refers to the actual sheets of paper. We start with the parameters related to pages, which are all dimen parameters. First there are \backslash htotal and \backslash vtotal , which should be equal to the horizontal and vertical size of the small pages you want. For instance, a pocket book has $\text{\backslash htotal} \approx 10.5$ cm, $\text{\backslash vtotal} \approx 17$ cm. Then there are $\text{\backslash horigin}$ and $\text{\backslash vorigin}$ which describe the "natural" offset of the text area from the top left corner of the page. These offsets are added to the \TeX offsets $\text{\backslash hoffset}$ and $\text{\backslash voffset}$. You may wonder why there are four parameters to describe the offsets, \TeX itself only has the latter two. The reason is that, unfortunately, \TeX has built in extra offsets of 1 inch. That may be right for American "letter" size paper, but on smaller pages it is generally too much. To make scaling of these offsets possible we had to introduce two extra parameters for them.

The names of parameters related to sheets all start with "sh"; all but one are dimen parameters. *Quire.tex* draws a number of lines to denote where the actual pages are on the sheet. As in the previous figure, lines are drawn around pairs of pages. The width is $\text{\backslash shoutline}$, so you can make them disappear with $\text{\backslash shoutline}=\text{0pt}$. The two pages are separated by two "staples," with width $\text{\backslash shstaplewidth}$ and length $\text{\backslash shstaplelength}$. You can make them disappear by making the width 0pt ; setting the length to $0.5\text{\backslash vtotal}$ will give you a solid line instead of two staples. The outlines are drawn exactly around the pages, the staples on top of the two pages. The widths of these lines does not influence the width of the pages. Finally, *quire.tex* draws "crop marks" of width \backslash shcrop . Crop marks are lines that indicate the outlines and center of the pages, without actually touching the pages. When you are cutting the paper, crop marks are sometimes more convenient than outlines. The crop marks are drawn so that, if they were extended, they would just be on the page. So they fall within the outlines that are drawn, which is easily observable if you make the widths excessive, say 1 cm.

The distance of the top of the sheet to the top of the pages is $\text{\backslash shvoffset}$ plus 1 inch. The pages are centered horizontally, therefore there is no parameter for horizontal offset. Under the pages, a footline is printed. The footline is determined by token register $\text{\backslash shfootline}$. By default the sheet number and the numbers of the two pages are printed (more about page numbers later), as well as the filename. The next paragraph explains the remaining four parameters, but if you are lucky you don't have to set them yourself:

They describe the printer you are using, so they need only be determined once. Your local installation may have macros that set these parameters automatically. Local macros are described at the end of this manual section.



In order to center the pages, `\shhtotal` must be set to the horizontal size of the sheets. The size is usually not a nice number, like 8.5 inch (216 mm), but more likely something like 220.1 mm. It is important to get this size right, since double sided printing requires properly centered pages. You probably have to experiment a bit with double sided pages to get this right, because often the printer introduces small inaccuracies. If a photocopier is used to turn single sided print into double sided print, you may need the parameters `\shhcorrection` and `\shvcorrection`. These denote extra offsets which are applied to odd numbered sheets only, to compensate for shifts introduced by “smart” copiers. `\shhcorrection` can sometimes be used to compensate for a wrong value of `\shhtotal`, but the author recommends that `\shhtotal` is set properly for the printer, and that the correction offsets are only used for copiers. If you take a stack of sheets and fold them, the outer sheets will have a longer way to go around the bend than the inner ones. In an attempt to compensate for that, the pages on each sheet are shifted outwards with respect to the pages on the next inner sheet, by an amount equal to `\shthickness`. Naturally, this parameter should have a rather small value.

Consider printing a quire of 20 pages. The first sheet that can be printed is sheet number 10 with pages 10 and 11. That means that \TeX has to store 10 complete pages in memory. If your \TeX version is small, there may not be enough memory for this, resulting in an error message

`TeX capacity exceeded, sorry.`

Luckily, there are some ways to reduce the number of pages that have to be stored. We already mentioned the use of `\shonly` to restrict output to odd or even sheets only. Doing that halves the number of pages that have to be stored. If that is not enough, you can use the macro `\makequire` instead of `\quire`. `\makequire` takes three arguments. The first argument is the same as for `\quire`, namely the number of pages per quire. Let s and r be the second and third arguments, respectively. Let n be the number of a sheet in the first quire. Then that sheet is only printed if $s \leq n < s + r$ (and similarly for following quires). This means that at most r pages need to be stored (or about half that if you also use `\shonly`). As an example, our 20 page quire can be printed in three runs with, respectively,

```
\makequire{20}{1}{4}
\makequire{20}{5}{4}
\makequire{20}{9}{4}
```

The runs result in sheets 4...1, 8...5, and 10...9. In the last run, the range r is larger than the actual number of sheets, but that causes no problems. Also, sheet s need not exist. In fact, `\quire{20}` is just an abbreviation for `\makequire{20}{0}{1000}`.

For previewing it is often inconvenient to use `\quire`, because of the reordering of pages. But you may still want to see how your text fits on the smaller pages. There are two macros which you can use instead of `\quire`. `\qtwopages` prints two pages per sheet, just like `quire`, but does not reorder the pages. All parameters mentioned before remain valid, except for `\shthickness`. Of course, you can also use this if you are not previewing a quire. For instance, if you have appropriately scaled fonts and you print in landscape mode, the following preamble will print your file two pages per sheet.


```
\input quire
\mag=500 % works in LaTeX too
\qtwopages
```


Finally, there is `\qonepage`, which prints just one page per sheet as usual. It does however draw outlines (no staples) and crop marks, and `\shonly` can also be used.

We claimed that `quire.tex` would work together with most other macro packages. But there is one type of macro that doesn't go well with quires, and usually you will have to switch them off when you are typesetting a quire. \TeX 's primitives for file handling, such as `\write`, are only performed when a page is actually shipped out. That is often used for the creation of an index or a contents page, in order to get the page numbers right. But `quire.tex` prints two pages per sheet, and a sheet has only one “page number,” so that such methods will most likely get all the page numbers wrong. Worse, the pages are shipped out in an unusual order, so that for instance a `\write` may take place before the file has been opened. Therefore,

you should avoid generation of an index or contents page when you are using `\quire`. In \LaTeX this is done by inserting `\nofiles` in the preamble. (For all but the final run you can use `\gonepage`, which does not change the page numbers.) Of course, if you want your booklet to start with a contents page, then that page has to be generated before the final \TeX run, since you cannot just rearrange the printed pages.

In `\shfootline`, `\pageno` refers to the sheet number (if you are using plain \TeX 's definitions); `\firstpageno` is the number of the first of the two pages (i.e., the page that was finished first, that can be either the left-hand or the right-hand page), and `\secondpageno` is the number of the other page. These three numbers are also shipped to the `dvi` file, as `\count0`, `\count1`, and `\count2`, respectively. When a page is finished but not yet printed on a sheet, the page number is displayed on your terminal between parentheses, as in `[(1)]`. When sheet number 7 with pages 11 and 14 is finished, `[7.11.14]` is displayed.

 `Quire.tex` does its job by redefining `\shipout`. Since `\shipout` is a \TeX primitive, such redefinition is not recommended; normally one would redefine the output routine instead. However, important parts of the typesetting of a text are often done in the output routine, which may well be different for different pages of a book. Since `quire.tex` is supposed to handle the whole book, and since it only handles complete pages without changing their layout, it seemed that in this particular case redefinition of `\shipout` was justified. The new `\shipout` can be used in the same way as the original one, except that you should not try to ship out a void box (which is a pretty meaningless thing to do anyway). Doing that will most likely result in a, seemingly unrelated, error message somewhere later in the text. The dangers of using `\openout`, `\closeout` and `\write`, without `\immediate`, were already explained. Like these file operations, `\special` works with “whatsits” which are handled when the page is shipped out. But as the argument of `\special` is expanded immediately, rather than when the page is shipped out, this is less likely to cause problems.

 `Quire.tex` uses mostly \TeX 's primitives, but also a few macros defined in `plain.tex`. The latter are `\newbox`, `\newcount`, `\newdimen`, `\newif`, `\newtoks`, `\nointerlineskip`, and `\offinterlineskip`, none of which are likely to be redefined by other macros. As mentioned before, `quire.tex` relies on \TeX 's endgame to generate complete quires. Although the endgame is built in, a really devious macro package may disable it. This is in fact exactly what happens in \LaTeX 's `\end` command, which is why we need `\latexquire` (which inserts an implicit `\endquire` at the end of the text).

1. *Printer definitions.* It is convenient if the parameters that depend (only) on the printer are predefined. As an example, here are some of the local definitions for the author's installation. When someone wants to use the printer called “maser,” all he or she has to do to set up those parameters is to say `\Pmaser` or `\Pmaserlandscape`. Note that this does not put any information in the `dvi` file, the user is still responsible for printing on the correct output device. There are similar definitions for other available printers. The next subsection lists the local definitions for your installation, if any.

```
\def\Pmaser
  {\shhtotal=220.02true mm
   \shthickness=0.21true mm
   \message{maser: flip paper so that short sides remain in place, %
    align closest right-hand corner in tray.}%
  }

\def\Pmaserlandscape
  {\shhtotal=281true mm
   \shthickness=0.21true mm
   \message{maser landscape: ...}
  }
```

2. *Local Caltech definitions.* The following macros will set the appropriate printer parameters: `\Pmaser`, `\Pmaserlandscape`, `\Plaser`, `\Plaserlandscape`, `\Phaser`, `\Phaserlandscape`, `\canon` (for the copier). For double-sided printing on maser, flip the pages so that the short edges remain in place. Align the closest right-hand corner in the tray. But for printing in landscape mode, flip the pages so that the long edges remain in place. On haser, you should rotate the pages 180°, but keep the text side up. You have to reorder the pages (or print the second time with `gtex -t`). Haser seems to have problems keeping the paper straight, which makes double-sided printing hard.

gloss.tex

Macros for vertically aligning words in consecutive sentences.
Version: 1.0 (26 November 1990) (doc: 10 November 1990)
Copyright © 1991 Marcel R. van der Goot

To gloss means to insert explanations or translations of words between the lines of a text. For ease of reading, one generally wants the explained word and the explanation to line up, as in

```
Dit is een Nederlands voorbeeld.
This is a Dutch example.
```

The macros defined in `gloss.tex` make it convenient to typeset such glossed sentences. To use the macros, you first have to `\input gloss*`. The main macro for typesetting glossed sentences is `\gloss`. It should be followed by the two sentences, each terminated by a `\return`. The previous example was typeset with

```
\gloss Dit is een Nederlands voorbeeld.
This is a Dutch example.
```

But instead of printing the glossed sentence, `\gloss` puts the whole construction in an `hbox`, called `\gline`. To actually insert that box in your text, you use `\box\gline` or `\unhbox\gline`. The text is put in a box to make it easier to handle. For instance, it makes it easy to put extra indentation or an example number in front of the sentence. (The baseline of the `hbox` is the baseline of the second line in the box.) The difference between `\box\gline` and `\unhbox\gline` is that the latter allows line breaks in the glossed sentence. That is obviously only useful if the glossed sentence is quite long; we will see an example later on.

The `\gloss` macro splits each sentence into words, and then aligns the words. In this context, a word is a sequence of characters separated by spaces. The number of spaces between words is irrelevant, so that you can line up the words in your input as well. Using

```
\gloss Dit is een Nederlands voorbeeld.
      This is a Dutch example.
```

gives exactly the same output as the input given before. However, you should not insert extra `\return` symbols. The sentences can have a different number of words. Sometimes one of the two sentences has a word without equivalent in the other sentence. In such cases you can insert an empty word with `{}`. You can also combine several words into a single word, by using grouping. The following example illustrates both.

```
\gloss Dit is een voorbeeldje in het Nederlands.
      This is a {little example} in {} Dutch.
```

gives

```
Dit is een voorbeeldje in het Nederlands.
This is a little example in Dutch.
```

Without the grouping “example” would have been lined up with “*in.*”

By default the first sentence of a glossed pair is set in italic font (`\it`), the second in roman font (`\rm`). You can change this by, for instance, setting `\let\eachwordone=\rm`, which will set each word in the first sentence in roman font. The font of the second sentence is governed by `\eachwordtwo`. This use of the `\eachword` macros is an instance of a more general mechanism that is explained later. You can insert font changes in the middle of a sentence, but they only apply to the next word.

```
\gloss Dit is een Nederlands voorbeeld.
      This is a \sl Dutch example.
```

will set “Dutch” in slanted font, but “example” in the default font. The `\gloss` macro considers `\sl Dutch` to be a single word, and sets this word in an `hbox` so that the effect of the font change is limited. `\sl` is not considered a word by itself, because spaces following a macro are always skipped.

* We have made no attempt to \LaTeX ify `gloss.tex`, but you can use the macros in \LaTeX just as in plain \TeX .

You may want to gloss more than a single sentence, maybe a whole paragraph. It would however be quite inconvenient if you had to follow `\gloss` by a whole paragraph on a single line (by terminating each line with a `%`-sign), followed by a line containing all the glosses. Especially if there are a lot of unglossed (empty) words, it would be hard to get the sentences lined up properly. Fortunately, a paragraph like the current one can be set quite easily. `\gloss` puts the created sentence in `\gline`, and `\gline` is emptied by using `\box` or `\unhbox`. But if you don't empty `\gline`, the next usage of `\gloss` will append the new sentence to the older one. Hence, this paragraph was typeset with

```
\gloss You may want to gloss more than a single sentence, maybe
      {} verb verb {} verb {} {} {} {} noun
\gloss a whole paragraph.
      {} {} noun
      ⋮ (10 more)
\gloss one. Hence, this paragraph was typeset with
      noun {} {} noun verb verb
\unhbox\gline
```

Of course, a few more things, like the fonts, had to be changed. We discuss that next. Also notice that it takes \TeX quite some time to typeset such a paragraph.



Each word from the first line following `\gloss` is set in an hbox as follows

```
\hbox{\eachwordone\strut<word>_}
```

The macro `\eachwordone` is by default `\let` equal to `\it`, but can be redefined. The previous paragraph was made with

```
\let\eachwordone=\rm
\def\sixstrut{\vrule height5pt depth1.5pt width0pt }
\def\eachwordtwo{\let\strut=\sixstrut\sixrm}
```

(If `\strut` were not redefined, there would be too much space between the first and the second line.) Furthermore, to obtain a larger separation between the pairs of lines, the `\baselineskip` was set to 22.5 pt (`\baselineskip` governs the spacing between the glossed pairs, the spacing within the pair is governed by the `\struts`.) Since a word does not contain spaces, and each word is followed by a space, the `\eachword` macros can easily be made to take the word (and the `\strut`) as argument. For instance, the following definition will underline every word in the first line.

```
\def\eachwordone#1_{\rm\underbar{#1}_}
```

The space following the word does not disappear at a line break, because it is hidden within an hbox. To hide this extra space between the last word and the right margin, the final thing needed to typeset the previous paragraph was

```
\setbox0=\hbox{ }\advance\rightskip by-\wd0
```

This is not perfect, because the size of the space depends on the font: there is a verb sticking out slightly.



The word boxes are paired together in a single vbox, and `\gline` consists of a sequence of these vboxes. The vboxes are separated by some glue, equal to `\glossglue`. Without this glue no line breaks would be possible, nor would the spaces between the words be stretchable. `gloss.tex` says `\glossglue=0pt plus2pt minus1pt`. The stretch and shrink components are only relevant if `\unhbox\gline` is used.

Sometimes you would rather not have extra space following a word, for instance when parts of a word are glossed separately: In

Dit is een voorbeeld- je in het Nederlands.
 This is an example (“little”) in Dutch.

it is preferable to have no space following “*voorbeeld-*” if it is not necessary. This can easily be achieved with a macro like

```
\def\+#1_{#1}
```

which is inserted just before the word that should not have a space: `\+voorbeeld- je`. It can also be done by making the hyphen gobble up the space, as in

```
\let\hyphen=-
\catcode'\active
\def-#1_{\hyphen#1}
```

However, this does not work if there is a word with a hyphen in the middle. To handle that situation, a more sophisticated definition of `-` can check what the next character is, using `\futurelet`. We will not do that here, as it has little to do with glossed sentences.

loop.tex

A simple looping construct (meta-macros).
 Version: 1.0 (15 April 1991) (doc: 15 Apr 1991)
 Copyright © 1991 Marcel R. van der Goot

The macros in `loop.tex` are intended to help you write your own macros, rather than to typeset text directly. Consequently, it is assumed that you have some experience writing macros; In other words, assume that this whole section is preceded by a “dangerous bend” symbol. These macros are kept simple and are not very robust: if you decide to use them in a different way than described below you are entirely responsible for what happens¹.

`Loop.tex` defines an alternative to plain \TeX 's `\loop` construct. (It can also be used with, e.g., \LaTeX .) Plain \TeX 's `\loop` is often awkward to use because of the limited form of tests in \TeX . The new construct tries to alleviate this problem by allowing a more flexible use of tests in the loop. To execute a loop, you say `\Loop ... \Pool`². This will normally execute the loop ‘body’, i.e., the commands corresponding to the dots, infinitely many times. There are two special commands you can use within the loop, `\Break` and `\Continue`. When a `\Break` command is executed, the rest of the body is ignored, and the loop is terminated. Execution of a `\Continue` command also causes the rest of the body to be ignored, but then the loop is started again from the beginning.

Both `\Break` and `\Continue` must be followed by a number. This number should be the number of `\if...` constructs that are terminated by the command. In other words, it is the number of `\fi`'s that has to be ‘executed’ to get proper balancing of all `\if...` tests. It is somewhat unfortunate that this number has to be supplied, but there seems to be no way to close all `\if...` tests automatically. Here is an example similar to one from the \TeX book.

```
\def\yes{Yes } \def\no{No }
\Loop\message{Are you happy? }
  \read-1 to\answer
  \ifx\answer\yes \Break1
  \else\ifx\answer\no \Continue2
  \fi \fi
  \message{(Please type Yes or No.)}
\Pool
```

At the position of `\Break` the nesting level of `\if...` tests is 1, at the position of `\Continue` the level is 2. If you forget the number you will (usually) get a ‘missing number’ error message, but possibly at an unexpected point in the program. The wrong number typically results in either an ‘Extra `\fi`’ or an ‘`\if... was incomplete`’ message.

There can be any number of `\Break` and `\Continue` commands in a loop, although it is wise to have at least one `\Break` command. The text following any of these commands up till the `\Pool` at the end of the loop must have balanced braces, and the loop body itself must also have balanced braces. If this causes a problem, you can use `\bgroup` and `\egroup` instead of `{` and `}`. If you want to nest two loops, you will have to enclose the inner loop within a group. (Incidentally, these restrictions on balancing braces and nesting loops also apply to plain \TeX 's `\loop` construct.)

¹ Not that the author assumes any responsibility even if you *do* obey the rules.

² We use initial capital letters to prevent confusion with plain \TeX 's `\loop`. Although you don't need `\loop` anymore, there may still be macros that depend on it (e.g., plain \TeX 's tabbing macros).

dolines.tex

Meta-macros to separate arguments by newlines and by empty lines.

Version: 1.0 (15 April 1991) (doc: 15 Apr 1991)

Copyright © 1991 Marcel R. van der Goot

The macros in *dolines.tex* are intended to help you write your own macros, rather than to typeset text directly. Consequently, it is assumed that you have some experience writing macros; in other words, assume that this whole section is preceded by a “dangerous bend” symbol.

*Dolines.tex** helps to write macros for which line breaks inserted by the user are important. When the definitions from *dolines.tex* are in effect, text consists of paragraphs separated by one or more empty lines (as usual). Each paragraph consist of one or more lines, where a line is terminated by a normal newline character, ‘ \sim ’ (normally, \TeX treats newlines as if they are spaces). Note that, as far as *dolines.tex* is concerned, a paragraph can only be terminated by an empty line, not by, e.g., `\par` or `\vskip`. To use *dolines.tex*, you have to define three macros, `\beforelines`, `\everyline#1`, and `\afterlines`. When *dolines.tex* detects the beginning of a paragraph, `\beforelines` is called. Next, for every line, `\everyline` is called with as argument a single macro. That macro has as its expansion the line that was read. (Usually, this is almost the same as saying that `\everyline` is called with the line as argument.) After the last line of the paragraph, `\afterlines` is called. No output is generated, other than what is produced by your own macros.

There are two ways of using *dolines.tex*. One is to read lines from a file, the other is to take the lines from the current input file. If you say

```
\filedolines{file name}
```

dolines.tex will attempt to read the file, calling `\beforelines`, `\everyline`, and `\afterlines` as described above. The file is read inside a group, so you may need to use `\global` if you do assignments within, e.g., `\everyline`.

As an example, we define a “poetry environment” which reads a poem from a file, preserving the poet’s line breaks. We draw horizontal lines around the poem, and put every stanza in a `vbox` to prevent page breaks. (It is certainly possible to write better poetry environments, we are just illustrating *dolines.tex*.)

```
\input dolines
\def\readpoem#1{\medskip \hrule \filedolines{#1}\hrule \medskip}
\def\beforelines{\medskip \vbox\bgroup}
\def\everyline#1{\hbox{\kern\parindent \it #1}}
\def\afterlines{\egroup \medskip} % end of the vbox
```

If file “*epic.tex*” contains a poem, we can typeset it with `\readpoem{epic}`.

However, if you are typesetting several short rhymes, it is somewhat inconvenient to have each of them in a separate file. This is where the second method of using *dolines.tex* is useful. You have to define a macro `\enddolines`, the expansion of which should be the text you want to use to terminate *dolines.tex*’s effect. You also have to define a macro `\finishdolines`. When you say `\begindolines`, *dolines.tex* starts reading lines, calling `\everyline` etc. This continues until *dolines.tex* reads a line equal to the (top level) expansion of `\enddolines`. Then your macro `\finishdolines` is called, and \TeX resumes normal processing. `\begindolines` and the line equal to `\enddolines` form a group (as with `\filedolines`); `\finishdolines` is called *outside* this group.

Here is how we can extend the poetry environment with two control sequences `\beginpoem` and `\endpoem`. We keep the above definitions of `\beforelines` etc.

```
\def\beginpoem{\medskip \hrule \begindolines}
\def\enddolines{\endpoem}
\def\finishdolines{\hrule \medskip}
```

* To use *dolines.tex* you also need *Midnight/loop.tex*, which will be `\input` when you `\input dolines`.

Now we can typeset a poem as follows (the English stanza is from The \TeX book):

```
\beginpoem
Roses are red,
\quad Violets are blue;
Rhymes can be typeset
\quad With boxes and glue.

Blauw de viooltjes,
\quad Rood zijn de rozen;
Een rijm kan gezet
\quad Met plaksel en dozen.
\endpoem
```

Note that we never defined `\endpoem`, as it is never expanded. In fact, there is no reason to terminate the poem with a control sequence. If we define `\def\enddolines{***}`, three stars on a single line will play the rôle of `\endpoem` above.



There are a few things to be careful about. First, a line must have properly nested `{...}` groups. (Because a line is read as an argument delimited by a `<return>` character.) You should not use this to fold long lines, because that will insert an explicit `<return>` character with catcode 12 in the line. Second, spaces at the beginning of a line are skipped, unless you change the catcode of `'_'`. This means that, for instance, there can be spaces in front of `\endpoem`. (Spaces are skipped at the beginning of a line because at that point \TeX is in “state N” — see Chapter 8 of The \TeX book.) Third, when `\everyline` is called, the line that is its argument is already tokenized. This means that a catcode change within `\everyline` cannot effect this line. Furthermore, when `\beforelines` is called, the first line is already read and tokenized.



`\filedolines` and `\begindolines` treat lines in pretty much the same way. There is however a significant difference in the way comments are treated. If a line has a comment (i.e., it ends with `'%...'`), `\begindolines` will miss the `<return>` at the end of the line. This is normal for \TeX , it can be used to fold long lines. However, if you are using `\filedolines`, comments will be skipped, but lines are not concatenated: `\filedolines` will not miss the `<return>`! (This is because `\filedolines` uses \TeX 's `\read` primitive.) A rather obscure difference is that a line consisting of just `'\par'` is read as an empty line by `\filedolines`, whereas it is read as a non-empty line by `\begindolines`. If you want to use `\begindolines` to read from a file, you can say `\expandafter\begindolines\input`

It is possible for `\finishdolines` to take arguments, but they have to appear on the line following the line with the expansion of `\enddolines`. Suppose we change the poem environment by defining `\finishdolines` as

```
\def\finishdolines#1%
  {\line{\strut \hss --- #1\hss\hss}\hrule \medskip}
```

Hence, `\finishdolines` will print the author's name under the poem. The argument should be written on the line following `\endpoem`:

```
\beginpoem
...
\endpoem
{A.U. Thor}
```

Sometimes we want to use `dolines.tex` to read just one paragraph by making `\enddolines` equal to an empty line. For instance, a macro `\address` which reads an address terminated by an empty line. The problem with defining `\def\enddolines{}` is that if there is an empty line before the address (or even if the rest of the line after `\address` is empty), no lines will be read at all. To ensure that at least one paragraph is read, we use `\beforelines` to define `\enddolines`:

```
\def\beforelines{\normalbeforelines\def\enddolines{}}
\let\enddolines=\relax
```

Here, `\normalbeforelines` denotes whatever must be done at the beginning of the address. Initially `\enddolines` is equal to `\relax`, but as soon as the first non-empty line is found, `\enddolines` is redefined as an empty line. Following that, an empty line causes calls of `\afterlines` and `\finishdolines`. (A way to include an empty line in the address is to write a tie (“~”) on a line of its own.)



As a more complicated example, we design a simple verbatim environment. The verbatim environment treats horizontal tabs as spaces, and considers multiple empty lines equal to a single empty line. We first write macros `\setupverbatim` and `\finishverbatim`. The first sets up the character codes for verbatim printing, the second resets those character codes.

```
\def\setupverbatim
  {\medskip \hrule \bgroup
   \def\do##1{\catcode'\##1=12 }\dospecials
   \obeyspaces \tt
  }

\def\finishverbatim{\egroup \hrule \medskip}
{\obeyspaces\global\let_=\_}
```

Grouping is used to keep the catcode changes local. Plain TeX’s `\dospecials` macro applies the macro `\do` to all special characters (`\`, `$`, etc.), which, in this case, gives them all catcode 12 (“other”). But we don’t want catcode 12 for spaces, because the `\tt` font has a visible space character ‘`_`’. Therefore, spaces are made active with `\obeyspaces`, and in the last line we make an active space equal to `_`.

To start setting verbatim text, we use `\beginverbatim`; `\readverbatim` is used to typeset a whole file in verbatim mode. The macros for typesetting lines and paragraphs are very straightforward.

```
\def\beginverbatim{\setupverbatim\begindolines}
\let\finishdolines=\finishverbatim

\def\readverbatim#1{\setupverbatim\filedolines{#1}\finishverbatim}

\def\beforelines{\smallskip}
\def\everyline#1{\hbox{\strut #1}}
\def\afterlines{\smallskip}
```

Finally, we have to define `\enddolines`. We want to match `\beginverbatim` with `\endverbatim`, so we would like to do

```
\def\enddolines{\endverbatim}
```

The problem is that this makes `\enddolines` expand to the *control sequence* `\endverbatim`. But while we are in verbatim mode, no control sequences are recognized, because ‘`\`’ is not an escape character. What we really want is to make the expansion of `\enddolines` equal to the *string* “`\endverbatim`”. This is done by temporarily making ‘`\`’ a normal character and ‘`/`’ an escape character:

```
{\catcode'\/=0 /catcode'/\=12
 /gdef/enddolines{\endverbatim}
}
```

This completes the verbatim environment. Note that `\endverbatim` is only recognized if it appears on a line by its own and is not preceded by spaces. If you are not afraid of confusion, you can change the name `\finishverbatim` to `\endverbatim`.

labels.tex

Macros to print address labels and bulk letters.
 Version: 1.0 (24 October 1991) (doc: 10 Sep 1991)
 Copyright © 1991 Marcel R. van der Goot

The macros defined in `labels.tex` can be used to print address labels. Such labels can be printed on normal paper, if you want to cut them yourself, or on peel-off labels. (Peel-off labels are available in various sizes for most types of printers and photo-copiers.) In order to use the macros you always have to start with `\input labels`, which reads `labels.tex`*. We will first explain how to typeset labels, then how to print bulk letters.

There are two slightly different ways to print labels. The first one is to say

```
\beginlabels
...
\endlabels
```

where in place of ‘...’ you type the labels separated by empty lines. The second is to simply say

```
\labelfile{...}
```

where ‘...’ is the name of a file that contains the labels. In either case a label consists of several lines, and labels are separated by one or more empty lines. You end your file as usual, with `\bye`. `Labels.tex` does not specify which font to use, which in plain `TEX` means that by default a 10 points Roman font is used. However, for legibility it is usually better to use a sans-serif font, for instance by writing

```
\font\sf=cmss12
\sf \baselineskip=14pt
```

In order to typeset labels, you must define the size and other attributes of labels. (There are default values, but they are unlikely to suit your needs, especially if you want to print on peel-off labels.) These attributes must be set before you call `\beginlabels` or `\labelfile`; therefore, a typical input file has the form

```
\input labels
\vlbls=5 \hlbls=2
\vlblsize=5cm
\labelfile{customers.lbl}
\bye
```

where file ‘`customers.lbl`’ has the addresses that must be typeset. The assignments to `\vlbls`, `\hlbls`, and `\vlblsize` define certain attributes. Next we describe which attributes there are.

First you have to tell `TEX` how labels must appear on a page. `\vlbls` must denote the vertical number of labels per page, `\hlbls` the horizontal number. `\vfirst` specifies the amount of white space between the top of the page and the top of the first label; `\hfirst` is the amount of white space to the left of the first label. Dimensions `\vinter` and `\hinter` specify the white space between labels.

Then you have to specify the size of each label. `\vlblsize` and `\hlblsize` specify the vertical and horizontal sizes of a single label. `\vindent` is the white space between the top of the label and the baseline of the first line. (`\vindent` is similar to `TEX`’s `\topskip` parameter.) `\hindent` is the amount of white space between the left edge of a label and the text of the label.

The parameters mentioned above are sufficient to typeset labels. There are however a few special parameters that are sometimes useful. By default, there is no outline around the labels. That is suitable for printing on peel-off labels, but inconvenient for use with previewers or plain paper. You can set `\lbloutline` to the thickness of the lines you want around your labels, e.g., `\lbloutline=0.5pt`. The outlines are drawn at the outside of the label (i.e., in the space between the labels).

If there is too much text on a label, an error message is given; for reference, the message contains the input text corresponding to the label. A label with an error is not typeset. When you typeset labels,

* To use `labels.tex` you also need `Midnight/dolines.tex` and `Midnight/loop.tex`, which are input automatically.

by default a file ‘lblerror.tex’ is created, which has the text of all the labels that did not fit. This makes it convenient to process those labels again, e.g., with `\labelfile`. However, if you set `\erroraction=0` before you start typesetting labels, file ‘lblerror.tex’ is not created. Setting `\erroraction=1` does not only suppress generation of the error file, it also forces printing of the labels that have errors. Any other value of `\erroraction` causes the default behavior.



The error messages refer to the natural size of a label. If `\vindent` and `\hindent`, which are glue parameters, have shrink components, it is possible that the text will actually fit on the label. However, usually these indentations have no stretch or shrink components.



There are two token registers, `\beforelbl` and `\afterlbl`, that can be used to put additional text on each label. They contain \langle vertical material \rangle that is inserted just before and after the text of the label, respectively. By default, these sequences are empty. One can for instance specify

```
\beforelbl={\line{---Confidential---\hfil}\smallskip}
```

to put “—Confidential—” at the beginning of each address. *Note:* Since addresses are normally read from bottom to top, the post office prefers it if you don’t put text after the address, unless it is part of the address. E.g., using `\afterlbl={\noindent USA}` is ok, but use `\beforelbl` if you want to add an “attention:…” line to each label.



If you want to use the labels for something else than addresses, you may want to center the text on the label. Horizontal centering can be obtained with `\hindent=Opt plus1fil`. Note that this centers the text according to the widest line; it does not center each line separately. Vertical centering is a bit more complicated, because of the way T_EX stacks boxes vertically. The following centers the label text vertically:

```
\count255=\baselineskip
\vindent=\count255sp plus1fil
```

This works because the actual vertical indentation is `\vindent` minus `\baselineskip`, since T_EX automatically inserts `\baselineskip` glue.

Within a label, you can type ‘&’ and ‘#’ for ‘&’ and ‘#’. (In T_EX, you usually have to type ‘\&’ and ‘\#’ to get these symbols.) You can use macros in a label, for instance for font changes. A line of a label must contain balanced braces; a ‘%’ sign will concatenate the next line. (Usually, labels do not contain macros, nor braces, nor percentage-signs.) If for some reason you want an empty line in the middle of a label, write a ‘tie’ (‘~’) on that line. (The ~ is replaced by a space in the output.)

Address labels are often used to send an identical letter to each addressee. The command `\bulk` makes it easy to print the address on each letter. You need two files, one with all the addresses, and one with the standard letter. Say that they are called ‘addr.tex’ and ‘letter.tex’, respectively. The command

```
\bulk{addr}{letter}
```

will typeset a copy of the letter for each address. In the letter you can use the control sequence `\bulkaddress`. This stands for a vbox that contains the address to which the letter corresponds. (The vbox contains an hbox for each line of the address; parameters like `\vindent` and `\hlblsize` are not important when `\bulk` is used.) Since ‘letter.tex’ is processed many times, it is best to put any definitions that apply to all letters in the file that contains the `\bulk` command, rather than in the letter itself. Each letter is read within a group, so all non-global definitions and assignments in the letter are local to the letter. Also note that the vbox containing the address is typeset before the letter is read. The `\bulk` command does not generate address labels; for that you have to use `\labelfile{addr}`.

styledef.tex

styledef.tex: Macros to selectively input parts of a file.
Version: 1.0 (16 April 1991) (doc: 19 Sep 1991)
Copyright © 1991 Marcel R. van der Goot

Consider the following situation: You write a set of T_EX macros to typeset letters. These macros must set fonts, margins, offsets, etc. However, you want to use different styles for different types of letters. For instance, a short letter may need extra white space at the top to get a full page, whereas this is not needed for a letter several pages long. And a business letter has a standard header, with subject and references, whereas a small note to a friend only has a date. Probably the macros for the different styles have a lot in common, a reason to put them all in one file. But the different styles may have some conflicting definitions, of which you only want to include one. Styledef.tex* provides some macros to `\input` from a file only parts belonging to a particular “style.” (Partial inclusion can also be useful for other things than styles; see the end of this manual section for an example.)

Before we explain the macros from styledef.tex, let us look at two other solutions. First, say the only difference between styles “wide” and “narrow” is the width of the text. Then we can write macros `\widestyle` and `\narrowstyle` that set the appropriate width. As long as the differences between styles are small and easily parametrized, this method works fine. If there are many differences, however, the macros for setting the style soon get very large. Furthermore, it is hard too share parts of styles.

A second method is the one used by L^AT_EX. One makes one file “letter.sty” with all the common definitions for letters; an additional file is made for each style. L^AT_EX’s `\documentstyle` macro is then used to include the proper files. E.g., one could write

```
\documentstyle[business,narrow]{letter}
```

to write a letter with the “business” and “narrow” style options. This method allows sharing of common definitions, and also allows the combination of different style options. The disadvantage is that it soon leads to a very large number of files, many of them with only a few definitions.

With the macros in styledef.tex, you would use only one or two files. Typical would be to have one file, “letter.tex,” with all the common definitions for letters, and one file, “letter.style,” with all the different styles. Only some parts of the latter file are included, through the macros below. Alternatively, everything can be put in a single file. (Using two files has the advantage that “letter.tex” can be shared by several users, who each have their own personalized “letter.style.”) Styledef.tex is of course especially useful with plain T_EX, since that does not have an equivalent mechanism. However, the macros can also be used with L^AT_EX; later on we describe how to make the `\documentstyle` command interact with the styledef macros.

First we describe how to `\input` only parts of a file. In order to use any of these macros, you first have to `\input styledef`. Then you have to allocate one or more “style registers”: a style register called “`\myst`” is allocated with the command

```
\newstyle\myst
```

You can give a new value to a style register through the `\setstyle` command. It takes a style register and a list of styles as arguments. E.g.,

```
\setstyle\myst{business, narrow}
```

For this command, as for all others, a style list is a list of style names separated by spaces and/or commas; the whole list must be enclosed in braces. The style names themselves can be any names that are meaningful to the macro package you are using. The `\addstyle` command is used in the same way as `\setstyle`, but it adds one or more styles to the style register without removing the old contents. You can use `\the\myst` to get the contents of the style register, and style registers can be assigned to each other using ‘=’. The last operation involving style registers is `\readstyle`. It takes a style register and a file name: The command

```
\readstyle\myst{letter.style}
```

* You also need Midnight/dolines.tex and Midnight/loop.tex, which will be `\input` automatically.

will `\input` file `letter.style`, but includes only those parts of it that belong to one of the styles in register `\myst`. That is all that is needed to selectively `\input` a file. The other macros from `styledef.tex` and the remainder of this manual section describe how to indicate in a file like `letter.style` which parts belong to which styles. If you don't intend to create such files yourself, there is no need to read on.

There are only three commands that govern which parts of a file belong to a particular style: `\styledef`, `\negstyledef`, and `\endstyle`. The first two are followed by a brace-enclosed list of styles (but not by a style register). When a file is read with `\readstyle`, the file is just `\input` by T_EX. However, when in that file a command

```
\styledef{business, long}
```

is encountered, T_EX checks whether the style register that is used with `\readstyle` contains style “business” or style “long.” If at least one of them is contained in the register, T_EX continues to input the file. But if none of the styles match, T_EX skips lines of the file until an `\endstyle` command is encountered. `\negstyledef` has exactly the opposite effect: if any style matches, the file is skipped till the next `\endstyle` command; only if no style matches is this part of the file included. The `\endstyle` command must occur on a line by itself, with nothing, not even a comment, following it. The `\endstyle` command stops T_EX from skipping text; if it is encountered when T_EX is not skipping text, it is equivalent to `\relax`.

Text that is not enclosed by a `\styledef ... \endstyle` (or `\negstyledef` equivalent) is always included. `\styledef` commands can be used within each other's range, but they do not actually nest: when part of the text is skipped, skipping always stops at the first `\endstyle`; any intervening `\styledef` commands are also skipped. E.g.,

```
\styledef{business}
\negstyledef{narrow}
...
\endstyle
```

The text corresponding to ‘...’ (which does not include any `\endstyle`) is only included if the style register contains style “business” and does not contain “narrow.” If style “business” is not contained in the register, the `\negstyledef` command as well as the ‘...’ are skipped. As a catch-all you may want to put an `\endstyle` at the end of the included file: if you missed an `\endstyle` somewhere, this catch-all will at least restrict the effects to this file.



The `\readstyle` command uses an internal style register to store which styles must be included. Therefore, if a file which is being read with `\readstyle`, uses `\readstyle` to read yet another file, the original list of styles will be lost. Although this situation seldomly occurs, there is a command `\savereadstyle` which takes a style register as argument and saves the internal style list in this register. Of course, if the second file should be read with exactly the same style list, you should just use `\input` instead of `\readstyle`. To restore the original style list, you use `\setreadstyle`, which also takes a style register as argument. `\setreadstyle` can also be used instead of `\readstyle`, if you want to select certain parts of the current file rather than `\input` a different file.



Style names can be built from any characters, except for spaces, commas, dots, and some characters with a special meaning to T_EX (e.g., braces, hash signs (#), and percentage signs). Also, characters in a style name must have the same catcode in the `\styledef` command as in the `\setstyle` command. When a `\styledef` or `\negstyledef` command is encountered, the following conditions must hold: The only escape character (catcode 0) is ‘\’; the only grouping characters (catcodes 1 and 2) are ‘{’ and ‘}’; the only parameter character (catcode 6) is ‘#’; and the only comment character (catcode 14) is ‘%’. After the `\styledef` command these catcodes may be changed, as long as the conditions hold again when the next `\styledef` is encountered. These particular catcode assignments are so standard that this rule is unlikely to ever cause problems.

Suppose you have written a file `letter.tex`, which is to be used by several people. You also have a file `letter.style` with some standard styles; people can copy and change this file to suit their own needs. To make life easier for those who only want to use your standard styles, you may not want to force them to learn about `styledef.tex`. Instead, you can define a macro `\style` which always reads a file with name `letter.style`. Its usage would be:

```
\input letter
\style business, long, narrow.
...
```

so that users need not be concerned about style registers etc. Here is part of an appropriate letter.tex:

```
\input styledef
\newstyle\letterstyle

\def\style#1.% terminate the argument with a dot
  {\setstyle\letterstyle{#1}
   \readstyle\letterstyle{letter.style}
  }
```



Something similar can be done with L^AT_EX's `\documentstyle`, provided you know which styles can appear in letter.style. This is done as follows: first, rename letter.tex to letter.sty, so that L^AT_EX can read it. Then add the following to the end of that file:

```
\newstyle\lst
\def\ds@business{\addstyle\lst{business}}
\def\ds@narrow{\addstyle\lst{narrow}}
... % same for every style
\@options
\readstyle\lst{letter.style}
```

You can then write `\documentstyle[narrow]{letter}`. L^AT_EX takes this as a command to read letter.sty. The `\@options` command causes L^AT_EX to execute, for every option 'xyz,' the command `\ds@xyz`, if it exists. All options for which this command does not exist lead to the input of a file xyz.sty, so that standard L^AT_EX options can still be used. Note that letter.sty is here the main style file (only there can `\@options` be used); be warned that it is not easy to write a main style file that satisfies all of L^AT_EX's assumptions.

Although we have suggested that these macros are suitable to handle different "styles," selective inclusion can also be useful in contexts unrelated to styles. For instance, you may write a paper which has some details and speculations which are not relevant for every reader. You can write the paper like this:

```
... % relevant thoughts
\styledef{details}
... % messy details
\endstyle
\negstyledef{details}
Details are left as an exercise for the reader.
\endstyle
... % important stuff
\styledef{speculations}
... % wild fantasies
\endstyle
```

If you want to print a version with speculations but without details, you can start the file with

```
\newstyle\print
\setstyle\print{speculations}
\setreadstyle\print
...
```


border.tex

Macros to typeset borders.

Version: 1.0 (21 October 1991) (doc: 23 Oct 1991)

Copyright © 1991 Marcel R. van der Goot

The macros described in this manual section are used to typeset borders, such as the border around this paragraph. These borders are built from different “characters”: in this example from quarter circles for the corners and straight line segments for the edges, but, given an appropriate font, other borders can also be typeset. (Below we explain how a border font must be designed in order to be useful with these macros.) Before any of the macros described below can be used, you must `\input border.tex`.*

There are several ways of typesetting borders, depending on the characters that form the border. Macros such as `\border` create a vbox of specified height and width (the depth is always zero), containing only the border. Typesetting text within a border is done by creating a box with the text and superimposing it on the box with the border, as explained later. The types of border we will now describe differ in the way they compose characters to build a border; the choice of the characters is determined by another macro (`\borderelm`). The border around the previous paragraph was typeset with `\border`. It is built from eight different characters: ‘`⌒`’, ‘`—`’, and ‘`⌓`’ for the top; ‘`⌑`’ and ‘`⌐`’ for the center; and ‘`⌔`’, ‘`—`’, and ‘`⌕`’ for the bottom. `\border` takes two arguments, the height and the width of the border.

```
\border{2cm}{10cm}
```

creates a border like this:

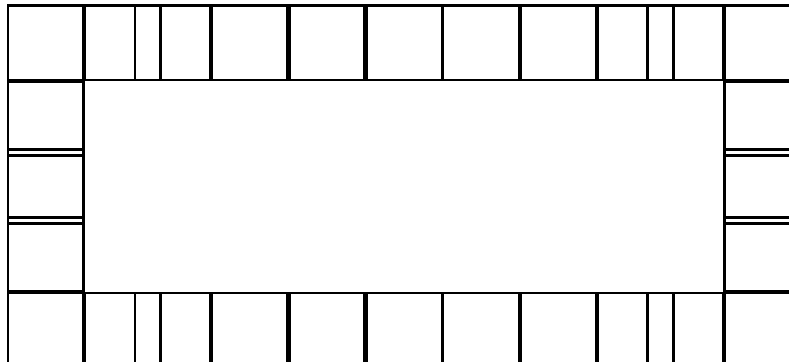


It is a vbox with height 2 cm, depth 0, and width 10 cm.

In order to create a border of the exact specified dimensions, some characters may need to overlap. Suppose we built a border from the following character, a 1 cm by 1 cm box:



The following is the border created by `\border{4.8cm}{10.5cm}`:




As you see, some of the boxes overlap, in order to get the specified dimensions. This overlap is always between the edge elements, there is never overlap with the corner elements (unless the size is too small). In

* Border.tex also works with L^AT_EX.

fact, the overlap is always with the edge elements that are closest to the corners. (`\border` always sets at least one edge element for each edge.)

`\border` works fine if the edge elements can overlap, which basically restricts them to be “rule like.” There is an alternative to `\border`, called `\xborder`. It is used in the same way as `\border`, but the exact size is obtained by putting extra space between the elements, rather than overlapping them. Suppose we build a border with the same corner elements as before, but with the edge built from “bullets” rather than straight line segments. The top element consists of a bullet centered in a 1 cm wide box. Here are the top lines obtained from `\border{...}{4.5cm}` and `\xborder{...}{4.5cm}`, respectively.

In case of `\xborder`, the proper vertical size is also obtained by spacing.

Some border characters, however, need to abut in order to form a border. For instance, suppose we want to make a “cloud like” border, where the top corners and top edge elements are all ‘’. In this case, `\border{...}{1.6cm}` and `\xborder{...}{1.8cm}` give

Clearly, neither solution works very well. For this case there are two other macros, `\minusborder` and `\plusborder`. They take the same arguments as `\border`, but produce a box of rounded dimensions rather than of the exact size. The border is built by abutting the elements. `\minusborder` rounds the size down, whereas `\plusborder` rounds the size up. `\minusborder{...}{1.8cm}` and `\plusborder{...}{1.8cm}` give


The above macros all generate empty borders. Typically, one wants to put some text in the border. There is one macro to make that easy, `\setborder`. It is called as follows:

```
\setborder[width, hspace, vspace]\border{ ... }
```

where the `...` stands for the text that must be boxed. In this case, `\border` will be used to create the border, but you can also use `\xborder` etc. The border will be `width` wide (or a rounded value); the height of the border is determined by the amount of text. The text itself is first broken in paragraphs and lines, as usual in \TeX . The width of the lines is `width - 2 · hspace`. The text is then put in the border with `vspace` space at the top and at the bottom, and `hspace` space at the left and at the right hand sides. For instance, the first paragraph of this manual section was set with

```
\setborder[\hsize,1em,\smallskipamount]\border
{\noindent The macros described ...
....
}
```

The `width` and `hspace` must be dimensions; `vspace` can be a dimension or a glue amount (without `\vskip`).

 With `\setborder`, the text between braces, `{...}`, is read in the same way as when you write `\vbox {...}`. (You can use `\bgroup` and `\egroup` instead of the braces.) The text is *not* an argument of `\setborder`, which is important if it contains catcode changes. You can change the paragraph shape, or have more than one paragraph of text.

For instance, the text in this paragraph is set with some extra `\leftskip` and `\rightskip` glue, which takes care of centering every line. This was done as follows:

```
\setborder[10cm, 1cm, 4mm]\border{
\parindent=0pt \parfillskip=0pt
\leftskip=0pt plus 1cm minus 5mm \rightskip=0pt plus 1cm minus 5mm
```

For instance, the text in this paragraph ...

```
...
}
```



If `\setborder` does not suit your purpose, you can write your own equivalent. Generally, to put text in a border, you first generate a box containing the border of the appropriate size. Then you make a box containing this border box plus the text superimposed on it. Superimposing is done by either “backspacing” using negative kerns, or by making a zero width or height border box, in a similar way as `\llap` makes a zero width box. Usually, you have to use several nested levels of boxing to get the desired result.

We have described the different ways of composing border characters, but not yet how to specify which characters are used. Whenever a border is typeset, it is assumed that a macro `\borderelm#1` exists. This macro is called with as argument a number in the range $0, \dots, 7$; it should return the appropriate border element, where the elements are numbered as follows:

```
0 1 1 1 1 1 1 1 1 1 2
3                                     4
3                                     4
5 6 6 6 6 6 6 6 6 6 7
```

`\borderelm` is called within an `hbox`; it should typeset a character, a box, or an `vrule`. (Since the macro is expanded within an `hbox`, you should be careful not to put extra spaces in it; doing that may cause strange gaps in the border.)

A convenient way of writing `\borderelm` is with an `\ifcase` construction. For instance, assume that in font `\qc` the characters `a`, `b`, `c`, etc. correspond to ‘`⌒`’, ‘`—`’, ‘`⌑`’, etc. Then you could write `\borderelm` as follows:

```
\def\borderelm#1%
  {\qc \ifcase#1
    a\or b\or c\or d\or e\or f\or g\or h% no extra space
  \fi
}
```


In fact, a macro like this is already defined in `border.tex`. It is called `\roundcorners`, so that you can say `\let\borderelm=\roundcorners` to get the borders with rounded corners. The corner elements come from font `\manfnt`, which corresponds to the external font ‘`manfnt`.’ (The corner elements are described on page 389 of *The T_EXbook*.) `\roundcorners` scales, which means that you can say, for instance,

```
\font\manfnt=manfnt scaled 2074
```

to get larger corners — provided you have the appropriately scaled font file (2074 corresponds to `\magstep4`). If you do not want to load the `manfnt` font (e.g., because your installation does not have it), you should define `\let\manfnt=\relax` before you `\input border.tex`.

Hopefully a variety of border fonts will come available in the future. But even with just the quarter circles you can already vary your borders quite a bit. We already saw a border with half circles before, it is an easy exercise to define a complete border that way. The following definition defines a fancier corner:

```
\setbox0=\hbox{\roundcorners2}
\ vbox{\offinterlineskip
  \hbox{\copy0\roundcorners0\copy0}
  \hbox to 3\wd0{\hfil\roundcorners7}
  \hbox to 3\wd0{\hfil\copy0}
}%
```

It defines the “number 2” (or number 5) corner: ; the other corners can be defined similarly.



If you want, you can omit the corners from `\borderelm` (i.e., write `\borderelm` to return nothing for arguments 0, 2, 5, and 7). But if you do not want any edge elements, you should still put something with width (or height) for the top/bottom (or left/right) edges; for instance, `\hbox to10pt{}` is a valid top element. `Border.tex` uses several box and dimen registers. Rather than reserving private registers, which would make them unavailable for other purposes, `border.tex` uses `\box0`, `\box2`, `\box4`, `\dimen0`, `\dimen2`, and `\count2`. Normally, this does not cause problems, since they are always restored to their old value. The only time you should take this into account, is in `\borderelm`: this macro should not depend on any global values of these registers, nor should it make any lasting changes to them.



If you are a font designer, and want to define a font that can be used with `border.tex`, you should understand something of how `\border` etc. build the borders. A border is built from three horizontal strips. The top and bottom strips consist of a corner element, a number of edge elements, and another corner element. These elements are aligned by their baselines, and the width of the border(-strip) is determined by the width of the bounding boxes. Therefore, the “ink” in the border elements should be close to the outside edges of the bounding boxes; otherwise there will not be a proper relation between the actual size of the border and the size of the visible border. The center strip is built from an edge, white space (glue), and another edge. The amount of white space is chosen so that the width of the center strip is the width of the top strip. The whole border is built by vertically aligning the top strip, a number of center strips, and the bottom strip. This is done by aligning the left hand edges of the horizontal strips. The vertical size of the border is again determined by the vertical size of the bounding boxes. You may notice that the quarter circle elements described on p. 389 of *The T_EXbook* (used in `\roundcorners`), do not fit properly in their bounding boxes for the purpose of setting borders. In fact, `\roundcorners` must go to considerable effort to reposition the bounding boxes. If you can, avoid such problems with any new fonts you design.

There are two more variants of borders. First, to set only one horizontal strip of a border generated by `\border`, you can use `\hborder`. It takes four argument: the width, and the numbers of the left element, middle element, and right element. E.g., `\border{4cm}012` generates the top of the border, whereas `\border{4cm}567` generates the bottom. There are also `\hxborder`, `\hminusborder`, and `\hplusborder`. However, there are no vertical equivalents.

Finally, there are macros `\minusgray` and `\plusgray`. They behave like their border counterparts, except that the center of the border is filled with an extra character (`\borderelm8`) rather than with white space. For use with these macros, `\borderelm` must return characters as follows:

```
0 1 1 1 1 1 1 1 1 1 2
3 8 8 8 8 8 8 8 8 8 4
3 8 8 8 8 8 8 8 8 8 4
5 6 6 6 6 6 6 6 6 6 7
```

This is useful if you have characters that contain a “gray” pattern (a “stipple” or “Ben Day” pattern). Although a font with a gray pattern should not be hard to define, to the author’s knowledge there is currently no such font in the public domain. These gray macros can also be used with `\setborder`.