Interaction tools: `dialog.sty` and `menus.sty`

Michael Downes

November 3, 1994

Introduction

This article describes `dialog.sty` and `menus.sty`, which provide functions for printing messages or menus on screen and reading users' responses. The file `dialog.sty` contains basic message and input-reading functions; `menus.sty` takes `dialog.sty` for its base and uses some of its functions in defining more complex menu construction functions. These two files are set up in the form of LaTeX documentstyle option files, but in writing them I spent some extra effort to try to make them usable with PLAIN TeX or other common macro packages that include PLAIN TeX in their base, such as $\mathcal{AMS}$-TeX or EPLAIN.

The appendix describes `grabhedr.sty`, required by `dialog.sty`, which provides two useful file-handling features: (1) a command `\inputfwh` that when substituted for `\input` makes it possible to grab information such as file name, version, and date from standardized file headers in the style promoted by Nelson Beebe—and to grab it in the process of first inputting the file, as opposed to inputting the file twice, or `\read`ing the information separately (unreliable due to system-dependent differences in the equivalence of TeX's `\input` search path and `\openin` search path). And (2) functions `\localcatcodes` and `\restorecatcodes` that make it possible for `dialog.sty` (or any file) to manage internal catcode changes properly regardless of the surrounding context.

These files and a few others are combined in a suite of files that goes by the name of **dialogl**, available on the Internet by anonymous ftp from CTAN (Comprehensive TeX Archive Network), e.g., `ftp.shsu.edu` (USA), or `ftp.uni-stuttgart.de` (Europe). The file `listout.tex` is a utility for verbatim printing of plain text files, with reasonably good handling of overlong lines, tab characters, other nonprinting characters, etc. It uses `menus.sty` to present an elaborate menu system for changing options (like font size, line spacing, or how many spaces should be printed for a tab character).

Here's an example from the menu system of `listout.tex` to demonstrate the use of some features from `dialog.sty` and `menus.sty`. First, the menu that you would see if you wanted to change the font or lin

```
==========================================
  F  Change font
  S  Change font size
  L  Change line spacing

Current settings: typewriter 8 / 10.0pt.

    Q  Quit          X  Exit         ?  Hel
==========================================

Your choice?
```

Suppose you wanted to change line spacing to 9 case `L`) and then `9pt`, except that on your first a `9pe` instead of `9pt`. Here's what you would see o

```
Your choice? l
Desired line spacing [TeX units] ? 9pe
?---I don't understand "9pe".
Desired line spacing [TeX units] ? 9pt

* New line spacing: 9.0pt
```

Both lowercase `l` and capital `L` are acceptable r line spacing is checked to make sure it's a valid ing, the internalized version of the user's value is the entered value was read correctly.

Now here's how the above menu is program `\menuF` is constructed using `\fxmenu`:

```
\fxmenu\menuF{}{
F  Change font
S  Change font size
```

```
L  Change line spacing
}{
Current settings: &\mainfont &\mainfontsize / %
&\the&\normalbaselineskip.
}
%
\def\moptionF{\lettermenu F}
```

In the definition of `\moptionF`, `\lettermenu` is a high-level function from `menus.sty` that prints `\menuF` on screen (given the argument `F`), reads a line of input from the user, extracts the first character and forces it to uppercase, then branches to the next menu as determined by that character. The response of `l` causes a branch to the function `\moptionFL`:

```
\def\moptionFL{%
  \promptmesj{%
    Desired line spacing [TeX units] ? }%
  \readline{Q}\reply
```

If `Q`, `X`, or `?` was entered, the test `\xoptiontest` will return 'true'; then we should skip the dimension checking and go directly to `\optionexec`, which knows what to do with those responses:

```
  \if\xoptiontest\reply
  \else
```

Otherwise we check the given dimension to make sure it's usable. If so, echo the new value as confirmation.

```
    \checkdimen\reply\dimen@
    \ifdim\dimen@>\z@
      \normalbaselineskip\dimen@\relax
      \normalbaselines
      \confirm{New line spacing:
        \the\normalbaselineskip}%
      \def\reply{Q}%
    \fi
```

If `\reply` was changed to `Q` during the above step, `\optionexec` will pop back up to the previous menu level (normal continuation); otherwise `\reply` retains its prior definition—e.g., `9pe`—to which `\optionexec` will simply say "I don't understand that" and repeat the current prompt.

```
  \fi
  \optionexec\reply
}
```

For maximum portability, `listout.tex` uses denominator ordinary printable ASCII characte menus can be obtained at a cost of forgoing sys using emTEX's `/o` option to output the box-draw DOS character set.

Notation

Double-hat notation such as `^^J` is used herein *TEXbook*, although occasionally the alternate fo emphasis is away from the character's tokenized breviations from `grabhedr.sty` are used freque `\expandafter`, and `\nx@` = `\noexpand`. Standa such as `\z@` or `\toks@` are used without special

Part 1
Basic dialog functions: `dialog.sty`

1.1 History

This file, `dialog.sty`, was born out of a utility c my personal use. The purpose of `listout.tex` text files—electronic mail, program source files i and, foremost, TEX macro files and log files. A gramming practice is to print out a macro file on ing corrections along the way, then use the mark file. (For one thing, this allows me to analyze the bus to work, or sitting out in the back yard put I normally desired was two 'pages' per shee landscape, in order to conserve paper.

Once created, `listout.tex` quickly became plain text files, not to mention an indispensable turn on `\tracingmacros` and `\tracingcommand` file so that I can see several hundred lines of the or three pages on my desk with 100+ lines per things out, label other things, draw arrows, and

I soon added a filename prompting loop to m files in a single run. In the process of perfectin over two or three years—and adding the ability number of columns at run time, eventually I w code that it became clear this code should be n own module. The result was `dialog.sty`.

Before getting into the macro definitions and technical commentary, here are descriptions from the user's perspective of the functions defined in this file.

1.2 Message-sending functions

`\mesj{`⟨*text*⟩`}`

Sends the message verbatim: category 12 for all special characters except braces, tab characters, and carriage returns:

```
{ } ^^I ^^M
```

Naturally, the catcode changes are effective only if `\mesj` is used directly, not inside a macro argument or definition replacement text.

Multiple spaces in the argument of `\mesj` print as multiple spaces on screen. A tab character produces always eight spaces; 'smart' handling of tabs is more complicated than I would care to attempt.

Line breaks in the argument of `\mesj` will produce line breaks on screen. That is, you don't need to enter a special sequence such as `^^J%` to get line breaks. See the technical commentary for `\mesjsetup` for details. Even though curly braces are left with their normal catcodes, they can be printed in a message without any problem, if they occur in balanced pairs. If not, the message should be sent using `\xmesj` instead of `\mesj`.

Because of its careful handling of the message text, `\mesj` is extremely easy to use. The only thing you have to worry about is having properly matched braces. Beyond that, you simply type everything exactly as you want it to appear on screen.

`\xmesj{`⟨*text*⟩`}`

This is like `\mesj` but expands embedded control sequences instead of printing them verbatim. All special characters have category 12 except backslash, percent, braces, tab, return, and ampersand:

```
\ % { } ^^I ^^M &
```

The first four have normal TEX catcodes to make it possible to use most normal TEX commands, and comments, in the message text. `^^I` and `^^M` are catcode 13 and behave as described for `\mesj`. The `&` is a special convenience, an abbreviation for `\noexpand`, to use for controlling expansion inside the message text.

Doubled backslash `\\` in the argument will produce a single category 12 backslash character—thus, `\\xxx` can be used instead of `\string\xxx` or `\noexpand\xxx` (notice that this works even for outer things like `\bye` or `\newif`). Similarly `\%`, `\{`, `\}` and `\&` produce the corresponding single characters.

Category 12 space means that you cannot write something like

```
\ifvmode h\else v\fi rule
```

in the argument of `\xmesj` without getting a spa
`\fi`.[1] Since occasionally this may be troublesor
ment of `\xmesj` to be a 'control word terminator
then `\foo\.xyz` produces `abcxyz` on screen (as
produce `abc xyz`). Thus the above conditional c

```
\ifvmode\.h\else\.v\fi\.rule
```

Even though the catcode changes done by `\xmes`
used inside an argument or definition replaceme
sionally to use `\xmesj` in those contexts, in order
setup. For instance, if you need to embed a messa
side a definition, you can write

```
\def\foo{...
  \xmesj{... this is a percent
    sign: \% (sans backslash) ...}
...}
```

To further support such uses of `\xmesj`, the fo
`\xmesj` setup: the backslash-space control symbo
`\Ω` and `\'` are defined to produce a `\newline`
duce a category-12 tilde.

Among other things, this setup makes it easi
spaces in an embedded message. For example, i
sage will have a line break on screen for each bac
third line will be indented four spaces.

```
\def\bar{...
  \xmesj{First line\
    Second line\
    \ \ \ \ Indented line\
    Last line}%
...}
```

The alternative of defining a separate message
and calling `\barfoo` inside of `\bar` would allow
lines and the multiple spaces, but would be slig
and hash table usage.

`\promptmesj{`⟨*text*⟩`}`
`\promptxmesj{`⟨*text*⟩`}`

These are like `\mesj`, `\xmesj` but use `\message`
internally, thus if the following operation is a `\re`

[1] Well, actually, you could replace each space by `%`⟨*newline*⟩ to get rid of it. But that makes the message text harder to read for the programmer.

screen at the end of the last line, as may be desired when prompting for a short reply, rather than at the beginning of the next line. The character ! is preempted internally for newlinechar, for these two functions only, which means that it cannot be actually printed in the message text. Use of a visible character such as !, rather than the normal \newlinechar ^^J, is necessary for robustness because of the fact that the \message primitive was unable to use an 'invisible' character (outside the range 32–126) for newlines up until TEX version 3.1415, which some users do not yet have (at the time of this writing—July 1994).

\storemesj\foo{⟨*text*⟩}
\storexmesj\foo{⟨*text*⟩}

These functions are similar to \mesj, \xmesj but store the given text in the control sequence \foo instead of immediately sending the message. Standard TEX parameter syntax can be used to make \foo a function with arguments, e.g. after

```
\storemesj\foo#1{...#1...}
```

then you can later write

```
\message{\foo{\the\hsize}}
```

and get the current value of \hsize into the middle of the message text. Consequently also in the x-version \storexmesj a category-12 # character can be obtained with \#.

\fmesj\foobar#1#2...{...#1...#2...}

Defines \foobar as a function that will take the given arguments, sow them into the message text {...}, and send the message. In the message text all special characters are category 12 except for braces, #, tab, and carriage return.

If an unmatched brace or a # must be printed in the message text \fxmesj must be used instead. (## could be used to insert a single category-6 # token into the message text, and TEX would print it without an error, but both \message and \write would print it as two ## characters, even though it's only a single token internally.)

\fxmesj\foobar#1#2...{...#1...#2...}

Combination of \xmesj and \fmesj. Defines \foobar like \fmesj, but with full expansion of the replacement text and with normal category codes for backslash, percent, braces, and hash #. The control symbols \\ \% \{ \} \& and \. can be used as in \xmesj, with also \# for printing a # character of category 12.

1.3 Reading functions

\readline{⟨*default*⟩}\answer

This reads a line of input from the user into the
can be anything chosen by the programmer, no
all special characters are deactivated, so that th
the user happens to enter something like \newif
the operating system, certain characters—e.g., CO
D, CONTROL-H—might have special effects instea
ment text of \answer, regardless of the catcod
vious example, under most operating systems, t
Backward-Delete key) will delete the previous cha
stead of entering an ASCII character 8 into \answ

There is one significant exception from the
\readline: spaces and tabs retain their norma
spaces in an answer will be reduced to a singl
space-delimited arguments will work when appli
any likely scenario where category 12 for spaces
code of ^^M is set to 9 (ignore) so that an empty
pressed the carriage return/enter key—will resul
swer is empty, the given default string will be su
be empty.

\xreadline{⟨*default*⟩}\answer

Like \readline but the answer is read as execu
the TEX special characters remain in effect whil
mon outer things (\bye, \+, \newif, ^^L, amo
the \read is done, but the user can still cause
outer control sequence or unbalanced braces. I c
tion, if the tokens are to remain executable, sho
the answer using \readline, writing it to a file,

\readchar{⟨*default*⟩}\answer

This is like \readline but it reduces the answer t
ther a single character or empty.

\readChar{⟨*default*⟩}\answer

This is like \readchar and also uppercases the a

\changecase\uppercase\answer

The function \changecase redefines its second a
to contain the same text as before, but uppercas
first argument. Thus \readChar{Q}\answer is e

```
\readchar{q}\answer
\changecase\uppercase\answer
```

It might sometimes be desirable to force lower case before using a file name given by the user, for example.

1.4 Checking functions

`\checkinteger\reply\tempcount`

To read in and check an answer that is supposed to be an integer, use `\readline\reply` and then apply `\checkinteger` to the `\reply`, supplying a count register, not necessarily named `\tempcount`, wherein `\checkinteger` will leave the validated integer. If `\reply` does not contain a valid integer the returned value will be `-\maxdimen`.

At the present time only decimal digits are handled; some valid TEX numbers such as `"AB`, `‘\@`, `\number\prevgraf`, or a count register name, will not be recognized by `\checkinteger`. There seems to be no bulletproof way to allow these possibilities.

Tests that hide `\checkinteger` under the hood, such as a `\nonnegativeinteger` test, are not provided because as often as not the number being prompted for will have to be tested to see if it falls inside a more specific range, such as 0–255 for an 8-bit number or 1–31 for a date, and it seems common sense to omit overhead if it would usually be redundant. It's easy enough to define such a test for yourself, if you want one.

`\checkdimen\reply\tempdim`

Analog of `\checkinteger` for dimension values. If `\reply` does not contain a valid dimension the value returned in `\tempdim` will be `-\maxdimen`.

Only explicit dimensions with decimal digits, optional decimal point and more decimal digits, followed by explicit units `pt cm in` or whatever are checked for; some valid TEX dimensions such as `\parindent`, `.3\baselineskip`, or `\fontdimen5\font` will not be recognized by `\checkdimen`.

What good is all this?

What good is all this stuff, practically speaking?—you may ask. Well, a typical application might be something like: At the beginning of a document, prompt interactively to find out if the user wants to print on A4 or US letter-size paper, or change the top or left margin. Such a query could be done like this:

```
\promptxmesj{
Do you want to print on A4 or US letter
Enter u or U for US letter, anything els
\readChar{A}\reply % default = A4
\if U\reply \textheight=11in \textwidth=
\else \textheight=297mm \textwidth=210mm
%    Subtract space for 1-inch margins
\addtolength{\textheight}{-2in}
\addtolength{\textwidth}{-2in}

\promptxmesj{
Left margin setting? [Return = keep curr
\the\oddsidemargin]: }
\readline{\the\oddsidemargin}\reply
\checkdimen\reply{\dimen0}
\ifdim\dimen0>-\maxdimen
  \setlength\oddsidemargin{\dimen0}%
  \xmesj{OK, using new left margin of %
\the\oddsidemargin.}
\else
  \xmesj{Sorry, I don't understand %
that reply: ‘\reply’.\
Using default value: \the\oddsidemargin
\fi
```

Although LATEX's `\typeout` and `\typein` functio they are rather more awkward, and checking the quite difficult.

1.5 Implementation

Standard package identification:

```
%<*2e>
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{dialog}[1994/11/08 v0.9y
%</2e>
```

1.6 Preliminaries

```
%<*2e>
\RequirePackage{grabhedr}
%</2e>
```

If grabhedr.sty is not already loaded, load it now. The \trap.input function is explained in grabhedr.doc.

```
%<*209>
\csname trap.input\endcsname
\input grabhedr.sty \relax
\fileversiondate{dialog.sty}{0.9y}{1994/11/08}%
%</209>
```

The functions \localcatcodes and \restorecatcodes are defined in grabhedr.sty. We use them to save and restore catcodes of any special characters needed in this file whose current catcodes might not be what we want them to be. Saving and restoring catcode of at-sign @ makes this file work equally well as a LaTeX documentstyle option or as a simple input file in other contexts. The double quote character " might be active for German and other languages. Saving and restoring tilde ~, hash #, caret ^, and left quote ‘ catcodes is normally redundant but reduces the number of assumptions we must rely on. (The following catcodes are assumed: \ 0, { 1, } 2, % 14, a–z A–Z 11, 0–9 . - 12. Also note that \endlinechar is assumed to have a non-null value.)

```
%% The line break is significant here:
\localcatcodes{\@{11}\ {10}\
{5}\~{13}\"{12}\#{6}\^{7}\‘{12}}
```

1.7 Definitions

For deactivating characters with special catcodes during \readline we use, instead of \dospecials, a more bulletproof (albeit slower) combination of \otherchars, \controlchars, and \highchars that covers all characters in the range 0–255 except letters and digits. Handling the characters above 127 triples the overhead done for each read operation or message definition but seems mandatory for maximum robustness.[2]

\otherchars includes the thirty-three nonalphanumeric visible characters (counting space as visible). It is intended as an executable list like \dospecials but, as an exercise in memory conservation, it is constructed without the \dos. For the usual application of changing catcodes, the list can still be executed nicely as shown below. Also, if we arrange to make sure that each character token gets category 12, it's not necessary to use control symbols such as \% in place of those few special characters that would otherwise be difficult to place inside of a definition. This avoids a problem that would otherwise arise if we included \+ in the list and tried to process the list with a typical definition of do: \+ is 'outer' in plain TeX and would cause an error message when \do attempted to read it as an argument. (As a matter of fact the catcode changes below show a different way around that problem, but a list of category-12 character tokens is a fun thing to have around anyway.)

```
\begingroup
```

First we start a group to localize \catcode cha
catcodes to 12 except for backslash, open brace,
dled by judicious application of \escapechar,
defining \do in a slightly backward way, so tha
don't need to worry about the presence of \+ i
tice the absence of \‘ from the list of control sy
12 in the \localcatcodes declaration at the b
would be troublesome to make the definition of \
bilities that ‘ might have catcode 0, 5, 9, or 14).

```
\def\do{12 \catcode‘}
\catcode‘\~\do\!\do\@\do\#\do\$\do\^\do\&
\do\*\do\(\do\)\do\-\do\_\do\=\do\[\do\]
\do\;\do\:\do\’\do\"\do\<\do\>\do\,\do\.
\do\/\do\?\do\|12\relax
```

To handle backslash and braces, we define \\,
sponding category-12 character tokens. Setting
\string will omit the leading backslash that it
plied to a control sequence.

```
\escapechar -1
\edef\\{\string\\}
\edef\{{\string\{}\edef\}{\string\}}
```

Space and percent are done last. Then, with al
category 12, it's rather easy to define \othercha

```
\catcode‘\ =12\catcode‘\%=12
\xdef\otherchars
{ !"#$%&’()*+,-./:;<=>?[\\]^_‘\{|\}~}
\endgroup %                ^     ^  ^
```

\controlchars is another list for the control ch
construction of this list is similar to the construc
turn off \endlinechar because the catcode of ^^
inside the \gdef is not a problem (as it might h
ness of ^^L) because the catcode is changed from

```
\begingroup
\endlinechar = -1
\def\do{12 \catcode‘}
\catcode‘\^^@\do\^^A\do\^^B\do\^^C
```

[2] If you are using dialog.sty functions on a slow computer, you might want to try setting \highchars = empty to see if that helps the speed.

```
\do\^^D\do\^^E\do\^^F\do\^^G\do\^^H\do\^^I
\do\^^J\do\^^K\do\^^L\do\^^M\do\^^N\do\^^O
\do\^^P\do\^^Q\do\^^R\do\^^S\do\^^T\do\^^U
\do\^^V\do\^^W\do\^^X\do\^^Y\do\^^Z\do\^^[
\do\^^\\do\^^]\do\^^^\do\^^_\do\^^? 12\relax
%
\gdef\controlchars{^^@^^A^^B^^C^^D^^E^^F^^G
  ^^H^^I^^J^^K^^L^^M^^N^^O^^P^^Q^^R^^S^^T
  ^^U^^V^^W^^X^^Y^^Z^^[^^\^^]^^^^^_^^?}
\endgroup
```

And finally, the list `\highchars` contains characters 128–255, the ones that have the eighth bit set.

```
\begingroup
\def\do{12 \catcode`}
\catcode`\^^80\do\^^81\do\^^82\do\^^83\do\^^84
\do\^^85\do\^^86\do\^^87\do\^^88\do\^^89\do\^^8a
\do\^^8b\do\^^8c\do\^^8d\do\^^8e\do\^^8f
\do\^^90\do\^^91\do\^^92\do\^^93\do\^^94\do\^^95

...

\do\^^fc\do\^^fd\do\^^fe\do\^^ff 12\relax
%
\gdef\highchars{%
^^80^^81^^82^^83^^84^^85^^86^^87^^88%
^^89^^8a^^8b^^8c^^8d^^8e^^8f%
^^90^^91^^92^^93^^94^^95^^96^^97^^98%

...

^^f9^^fa^^fb^^fc^^fd^^fe^^ff}
\endgroup
```

The function `\actively` makes a given character active and carries out the assignment given as the first argument. The assignment can be embedded in the replacement text of a macro definition without requiring any special setup to produce an active character in the replacement text. The argument should be a control symbol, e.g. `\@` or `\#` or `\'`, rather than a single character. (Except that `+` is safer than `\+` in PLAIN TEX.) If the assignment is a definition (`\def`, `\edef`, `\gdef`, `\xdef`) it is allowed to take arguments in the normal TEX way. Prefixes such as `\global`, `\long`, or `\outer` must go inside the first argument rather than before `\actively`.

Usage:

```
\actively\def\?{⟨replacement text⟩}
\actively\def\%#1#2{⟨replacement text⟩}
\actively{\global\let}\^^@=\space
```

One place where this function can be put to good
der to get special action at the end of each line
about this would be to write

```
\def\par{something}\obeylines
```

which is a puzzling construction to the TEX
`\obeylines` does with `\par`. The same effect co
parently with

```
\actively\def\^^M{something}
```

In the definition of `\actively` we use the
to create an active character with the right ch
`\begingroup \endgroup` structure that localizes

```
\def\actively#1#2{\catcode`#2\active
  \begingroup \lccode`\~=`#2\relax
  \lowercase{\endgroup#1~}}
```

The `\mesjsetup` function starts a group to localiz
be closed eventually by a separate function that
the message text for later retrieval.

We want to change the catcode of each charac
`\controlchars`, and `\highchars` to 12. After gi
apply it to each of the three lists, adding a suitab
make the recursion stop there. This allows leavin
acter lists, without incurring the cost of an if tes

```
\def\mesjsetup{\begingroup \count@=12
  \def\do##1{\catcode`##1\count@ \do}%
```

The abbreviation `\xp@` = `\expandafter` is from

```
  \xp@\do\otherchars{a11 \@gobbletwo}%
  \xp@\do\controlchars{a11 \@gobbletwo}%
  \xp@\do\highchars{a11 \@gobbletwo}%
```

Make the tab character produce eight spaces:

```
  \actively\edef\^^I{ \space\space\space
    \space\space\space\space}%
```

The convenient treatment of newlines in the arg
produces a line break on screen) is achieved by
and defining it to produce a category-12 `^^J` c
would have sufficed to make `^^M` category 12 an

while sending the message, it turns out to be useful for other functions to have the ^^M character active, so that it can be remapped to an arbitrary function for handling new lines (e.g., perhaps adding extra spaces at the beginning of each line). And if \mesj treats ^^M the same, we can arrange for it to share the setup routines needed for the other functions.

```
\endlinechar=‘\^^M\actively\let\^^M=\relax
\catcode‘\{=1 \catcode‘\}=2 }
```

In \sendmesj we go to a little extra trouble to make sure ^^M produces a newline character, no matter what the value of \newlinechar might be in the surrounding environment. The impending \endgroup will restore \newlinechar to its previous value. One reason for using ^^J (instead of, say, ^^M directly) is to allow e.g. \mesj{xxx^^Jxxxx} to be written inside a definition, as is sometimes convenient. This would be difficult with ^^M instead of ^^J because of catcodes.

```
\def\sendmesj{\newlinechar‘\^^J%
  \actively\def\^^M{^^J}%
  \immediate\write\sixt@@n{\mesjtext}\endgroup}
```

Given the support functions defined above, the definition of \mesj is easy: Use \mesjsetup to clear all special catcodes, then set up \sendmesj to be triggered by the next assignment, then read the following balanced-braces group into \mesjtext. As soon as the definition is completed, TEX will execute \sendmesj, which will send the text and close the group that was started in \mesjsetup to localize the catcode changes.

```
\def\mesj{\mesjsetup \afterassignment\sendmesj
  \def\mesjtext}
```

The \sendprompt function is just like \sendmesj except that it uses \message instead of \write, as might be desired when prompting for user input, so that the on-screen cursor stays on the same line as the prompt instead of hopping down to the beginning of the next line. In order for newlines to work with \message we must use a visible character instead of ^^J. When everyone has TEX version 3.1415 or later this will no longer be true. The choice of ! might be construed (if you wish) as editorial comment that ! should not be shouted at the user in a prompt.

```
\def\sendprompt{%
  \newlinechar‘\!\relax \actively\def\^^M{!}%
  \message{\mesjtext}\endgroup}
```

This function is like \mesj but uses \sendprompt instead of \sendmesj.

```
\def\promptmesj{\mesjsetup
  \afterassignment\sendprompt \def\mesjtext}
```

Arg #1 of \storemesj is the control sequence u
be stored.

```
\def\storemesj#1{\mesjsetup
  \catcode‘\#=6 % to allow arguments if r
  \afterassignment\endgroup
  \long\gdef#1}
```

While \storemesj\foo{...} is more or less the
cial catcode changes, \fmesj\foo{...} corresp
that is, after \fmesj\foo the function \foo ca
message. Thus \storemesj is typically used fo
\fmesj is used for storing entire messages.

To read the parameter text #2, we use the pec
erything up to the opening brace.

```
\def\fmesj#1#2#{\mesjsetup
  \catcode‘\#=6 % restore to normal
```

The parameter text #2 must be stored in a to
to avoid problems with # characters. The \lon
likely) possibility of using \fmesj to define so
saying ‘You can’t use #1 here’ where one of the p

```
  \toks@{\long\gdef#1#2}%
```

Define \@tempa to put together the first two ar
and make the definition of #1.

```
  \def\@tempa{%
    \edef\@tempa{%
      \the\toks@{%
```

The abbreviation \nx@ = \noexpand is from gra

```
        \begingroup\def\nx@\mesjtext{\the\t
          \nx@\sendmesj}%
    }%
    \@tempa
    \endgroup % Turn off the \mesjsetup c
  }%
  \afterassignment\@tempa
  \toks2=}
```

\xmesjsetup is like \mesjsetup except it prep
kens and normal comments in the message text.
tain other features are thrown in.

Here, unlike the setup for \xreadline, I d
ness of \bye, \newif, etc., because I presume th

\storexmesj, \fxmenu, etc. are more likely to be written by a TEXnician than by an average end user, whereas \xreadline is designed to handle arbitrary input from arbitrary users.

```
\def\xmesjsetup{\mesjsetup
```

Throw in pseudo braces just in case we are inside an \halign with \\ let equal to \cr at the time when \xmesjsetup is called. (As might happen in $\mathcal{AMS}$-TEX.)

```
  \iffalse{\fi
  \catcode`\\=0 \catcode`\%=14
```

Define \% \\ \{ \} \& to produce the corresponding single characters, category 12. The \lowercase trick here allows these definitions to be nonglobal.

```
  \begingroup \lccode`\0=`\\\lccode`\1=`\{%
  \lccode`\2=`\}\lccode`\3=`\%%
  \lowercase{\endgroup \def\\{0}\def\{{1}%
    \def\}{2}\def\%{3}}%
  \iffalse}\fi
  \edef\&{\string &}%
```

Let & = \noexpand for expansion control inside the argument text; let active ^^M = \relax so that newlines will remain inert during the expansion.

```
  \actively\let\&=\noexpand
  \actively\let\^^M=\relax
```

Define \. to be a noop, for terminating a control word when it is followed by letters and no space is wanted.

```
  \def\.{}%
```

Support for use of \xmesj inside a definition replacement text or macro argument: control-space \␣ = \space, tilde ~ prints as itself, \' (i.e., a lone backslash at the end of a line) will produce a newline, also \Ω, while finally \par = blank line translates to two newlines.

```
  \def\ { }\edef~{\string ~}%
```

Define \' to produce an active ^^M character, which (we hope) will be suitably defined to produce a newline or whatever.

```
  \begingroup \lccode`\~=`\^^M%
    \lowercase{\endgroup \def\^^M{~}}%
  \let\^^J\^^M \def\par{\^^M\^^M}%
}
```

\xmesj uses \xmesjsetup and \edef.

```
\def\xmesj{\xmesjsetup \afterassignment\sendmesj
  \edef\mesjtext}
```

\promptxmesj is analogous to \promptmesj, but

```
\def\promptxmesj{\xmesjsetup
  \afterassignment\sendprompt \edef\mesjt
```

And \storexmesj is like \storemesj, with exp for the function being defined, we also must defi 12 # character so that there will be a way to pri

```
\def\storexmesj#1#2#{\xmesjsetup
  \catcode`\#=6 % to allow arguments if r
  \edef\#{\string##}%
  \afterassignment\endgroup
  \long\xdef#1#2}
```

And \fxmesj is the expansive analog of \fmesj.

```
\def\fxmesj#1#2#{\xmesjsetup
  \catcode`\#=6 % restore to normal
  \edef\#{\string##}%
  \toks@{\long\xdef#1#2}%
  \def\@tempa{%
    \edef\@tempa{%
      \the\toks@{\begingroup
      \def\nx@\nx@\nx@\mesjtext{\the\toks
      \nx@\nx@\nx@\sendmesj}}%
    \@tempa % execute the constructed xde
    \endgroup % restore normal catcodes
  }%
  \afterassignment\@tempa
  \toks\tw@=}
```

1.8 Reading functions

The \readline function gets one line of input f default to be used if the user response is empty (turn/enter key), #2 macro to receive the input.

```
\def\readline#1#2{%
  \begingroup \count@ 12 %
  \def\do##1{\catcode`##1\count@ \do}%
  \xp@\do\otherchars{a11 \@gobbletwo}%
  \xp@\do\controlchars{a11 \@gobbletwo}%
  \xp@\do\highchars{a11 \@gobbletwo}%
```

Make spaces and tabs normal instead of category

```
  \catcode`\ =10 \catcode`\^^I=10 %
  \catcode`\^^M=9 % ignore
```

Reset end-of-line char to normal, just in case.

```
  \endlinechar`\^^M
```

We go to a little trouble to avoid \gdef-ing #2, in order to prevent save stack buildup if the user of \readline carries on unaware doing local redefinitions of #2 after the initial read.

```
  \read\m@ne to#2%
  \edef#2{\def\nx@#2{#2}}%
  \xp@\endgroup #2%
  \ifx\@empty#2\def#2{#1}\fi
}
```

\xreadline is like \readline except that it leaves almost all catcodes unchanged so that the return value is executable tokens instead of strictly character tokens of category 11 or 12.

```
\def\xreadline#1#2{%
  \begingroup
```

Render some outer control sequences innocuous.

```
    \xp@\let\csname bye\endcsname\relax
    \xp@\let\csname newif\endcsname\relax
    \xp@\let\csname newcount\endcsname\relax
    \xp@\let\csname newdimen\endcsname\relax
    \xp@\let\csname newskip\endcsname\relax
    \xp@\let\csname newmuskip\endcsname\relax
    \xp@\let\csname newtoks\endcsname\relax
    \xp@\let\csname newbox\endcsname\relax
    \xp@\let\csname newinsert\endcsname\relax
    \xp@\let\csname +\endcsname\relax
    \actively\let\^^L\relax
  \catcode`\^^M=9 % ignore
  \endlinechar`\^^M% reset to normal
  \read\m@ne to#2%
  \toks@\xp@{#2}%
  \edef\@tempa{\def\nx@#2{\the\toks@}}%
  \xp@\endgroup \@tempa
  \ifx\@empty#2\def#2{#1}\fi
}
```

\readchar reduces the user response to a single character.

```
\def\readchar#1#2{%
  \readline{#1}#2%
```

If the user's response and the default response a
after #1 to keep \@car from running away, so we

```
  \edef#2{\xp@\@car#2#1{}\@nil}%
}
```

\readChar reduces the user response to a single
ful to simplify testing the response later with \i

```
\def\readChar#1#2{%
  \readline{#1}#2%
  \changecase\uppercase#2%
```

Reduce #2 to its first character, or the first chara
tra braces {} are to prevent a runaway argument
both empty.

```
  \edef#2{\xp@\@car #2#1{}\@nil}%
}
```

The function \changecase uppercases or lowerca
ond argument, which must be a macro. The firs
or \lowercase.

```
\def\changecase#1#2{\@casetoks\xp@{#2}%
  \edef#2{#1{\def\nx@#2{\the\@casetoks}}
```

We allocate a token register just for the use of
used at a low level internally where we don't w
the scratch token registers 0–9.

```
\newtoks\@casetoks
```

A common task in reading user input is to v
kind was requested, that the response has indeed
a nonnegative integer is required for subsequent
ify that we have a nonnegative integer in hand bef
to inconvenient error messages. However, it's no
dle such verification. One possibility might be to

```
 \readnonnegativeinteger\foo
```

to do all the work of going out and fetching a
ing it in the macro \foo. Another possibility w
ing \readline and then apply a separate functio
with \if, for example

```
   \readline{}\reply
   \if\validnumber\reply ... \else ... \
```

For maximum flexibility, a slightly lower-level approach is chosen here. The target syntax is

```
\readline{}\reply
\checkinteger\reply\tempcount
```

where \tempcount will be set to -\maxdimen if \reply does *not* contain a valid integer. (Negative integers are allowed, as long as they are greater than -\maxdimen.) Then the function that calls \checkinteger is free to make additional checks on the range of the reply and give error messages tailored to the circumstances. And the handling of an empty \reply can be arbitrarily customized, something that would tend to be inconvenient for the first method mentioned.

The first and second approaches can be built on top of the third if desired, e.g. (for the second approach)

```
\def\validnumber#1{TT\fi
  \checkinteger#1\tempcount%
  \ifnum\tempcount>-\maxdimen }
```

The curious TT\fi...\ifnum construction is from TeXhax 1989, no. 20 and no. 38 (a suggestion of D. E. Knuth in reply to a query by S. von Bechtolsheim).

The arguments of \checkinteger's are: #2, a count register to hold the result; #1, a macro holding zero or more arbitrary characters of category 11 or 12.

```
\def\checkinteger#1#2{\let\scansign@\@empty
  \def\scanresult@{#2}%
  \xp@\scanint#1x\endscan}
```

To validate a number, the function \scanint must first scan away leading + or - signs (keeping track in \scansign@), then look at the first token after that: if it's a digit, fine, scan that digit and any succeeding digits into the given count register (\scanresult@), ending with \endscan to get rid of any following garbage tokens that might just possibly show up.

Typical usage of \scanint includes initializing \scansign@ to empty, as in the definition of \checkinteger.

```
\let\scansign@\@empty
\def\scanresult@{\tempcount}%
\xp@\scanint\reply x\endscan
```

Assumption: \reply is either empty or contains only category 11 or 12 characters (which it will if you used \readline!). If a separate check is done earlier to trap the case where \reply is empty—for example, by using a nonempty default for \readline—then the x before \endscan is superfluous.

Arg #1 = next character from the string bein
decimal digit is similar in spirit to the test \if!
(*The TEXbook*, Appendix D, p. 376).

```
\def\scanint#1{%
  \ifodd 0#11 %
```

Is #1 a decimal digit? If so read all digits into
fix.

```
    \def\@tempa{\afterassignment\endscan
      \scanresult@=\scansign@#1}%
  \else
    \if -#1\relax
```

Here we flipflop the sign; watch closely.

```
      \edef\scansign@{%
        \ifx\@empty\scansign@ -\fi}%
      \def\@tempa{\scanint}%
    \else
```

A plus sign can just be ignored.

```
      \if +#1\relax
        \def\@tempa{\scanint}%
      \else % not a valid number
        \def\@tempa{%
          \scanresult@=-\maxdimen\endscan
  \fi\fi\fi
  \@tempa
}
```

The \endscan function just gobbles any remain
as the argument delimiter.

```
\def\endscan#1\endscan{}
```

\dimenfirstpart, a count register, receives the
mal point. \dimentoks, a token register, receives

```
\newcount\dimenfirstpart
\newtoks\dimentoks
```

\scandimen is similar to \scanint but has to ca
the various subcomponents of a dimension (leadi
part, and units, with optional true, in addition t
ments of TEX's syntax for dimensions are a digi
the other components are optional (*The TEXboo*

When scanning for the digits of a fractional part, we can't throw away leading zeros; therefore we don't read the fractional part into a count register as we did for the digits before the decimal point; instead we read the digits one by one and store them in `\dimentoks`.

The function that calls `\scandimen` should initialize `\scansign@` to `\@empty`, `\dimenfirstpart` to `\z@`, `\dimentoks` to empty `{}`, and `\dimentrue@` to `\@empty`.

Test values: `0pt`, `1.1in`, `-2cm`, `.3mm`, `0.4dd`, `5.cc`, `.10000000009pc`, `\hsize`, `em`.

```
\def\scandimen#1{%
  \ifodd 0#11
    \def\@tempa{\def\@tempa{\scandimenb}%
      \afterassignment\@tempa
      \dimenfirstpart#1}%
  \else
```

The following test resolves to true if `#1` is either a period or a comma (both recognized by TEX as decimal point characters).

```
    \if \if,#1.\else#1\fi.%
      \def\@tempa{\scandimenc}%
    \else
      \if -#1% then flipflop the sign
        \edef\scansign@{%
          \ifx\@empty\scansign@ -\fi}%
        \def\@tempa{\scandimen}%
      \else
        \if +#1% then ignore it
          \def\@tempa{\scandimen}%
        \else % not a valid dimen
          \def\@tempa{%
            \scanresult@=-\maxdimen\endscan}%
  \fi\fi\fi\fi
  \@tempa
}
```

Scan for an optional decimal point.

```
\def\scandimenb#1{%
  \if \if,#1.\else#1\fi.%
    \def\@tempa{\scandimenc}%
  \else
```

If the decimal point is absent, we need to put back `#2` and rescan it to see if it is the first letter of the units.

```
    \def\@tempa{\scanunitsa#1}%
  \fi
  \@tempa
}
```

Scan for the fractional part: digits after the deci

```
\def\scandimenc#1{%
```

If `#1` is a digit, add it to `\dimentoks`.

```
  \ifodd 0#11 \dimentoks\xp@{%
      \the\dimentoks#1}%
    \def\@tempa{\scandimenc}%
  \else
```

Otherwise rescan `#1`, presumably the first letter

```
  \def\@tempa{\scanunitsa#1}%
  \fi
  \@tempa
}


\def\scanunitsa#1\endscan{%
```

Check for `true` qualifier.

```
  \def\@tempa##1true##2##3\@tempa{##2}%
```

The peculiar nature of `\lowercase` is evide to only the test part of the conditional without lems. (Compare the braces in this example to s `A}\else B}\fi`.)

```
  \lowercase{%
    \xp@\ifx\xp@\end
    \@tempa#1true\end\@tempa
  }%
```

No `true` was found:

```
    \let\dimentrue@\@empty
    \def\@tempa{\scanunitsb#1\endscan}%
  \else
    \def\dimentrue@{true}%
    \def\@tempa##1true##2\@tempa{%
      \def\@tempa{##1}%
      \ifx\@tempa\@empty
        \def\@tempa{\scanunitsb##2\endsc
      \else
        \def\@tempa{\scanunitsb xx\endsc
      \fi}%
```

```
    \@tempa#1\@tempa
  \fi
  \@tempa
}
```

Scan for the name of the units and complete the assignment of the scanned value to `\scanresult@`. Notice that, because of the way `\scanunitsb` picks up `#1` and `#2` as macro arguments, `p t` is allowed as a variation of `pt`. Eliminating this permissiveness doesn't seem worth the speed penalty that would be incurred in `\scanunitsb`.

The method for detecting a valid units string is to define the scratch function `\@tempa` to apply TEX's parameter-matching abilities to a special string that will yield a boolean value of true if and only if the given string is a valid TEX unit.

```
\def\scanunitsb#1#2{%
  \def\@tempa##1#1#2##2##3\@nil{##2}%
  \def\@tempb##1{T\@tempa
    pcTptTcmTccTemTexTinTmmTddTspT##1F\@nil}%
```

Force lowercase just in case the units were entered with uppercase letters (accepted by TEX, so we had better accept uppercase also).

```
  \lowercase{%
  \if\@tempb{#1#2}%
  }%
   \scanresult@=\scansign@
     \number\dimenfirstpart.\the\dimentoks
     \dimentrue@#1#2\relax
  \else
    \scanresult@=-\maxdimen
  \fi
```

Call `\endscan` to gobble garbage tokens, if any.

```
  \endscan
}
```

Argument `#2` must be a dimen register; `#1` is expected to be a macro holding zero or more arbitrary characters of category 11 or 12.

```
\def\checkdimen#1#2{%
  \let\scansign@\@empty \def\scanresult@{#2}%
  \let\dimentrue@\@empty
  \dimenfirstpart\z@ \dimentoks{}%
  \xp@\scandimen#1xx\endscan
}
```

Finish up.

```
\restorecatcodes
\endinput
```

Part 2
Menu functions: `menus.sty`

2.1 Function descriptions

```
\fmenu\foobar{
 ⟨preliminary text⟩
}{
 ⟨menu lines⟩
}{
 ⟨following text⟩
}
```

Defines `\foobar` as a function that puts the pre
of choices), and the after text on screen. Normal

```
\foobar           % print the menu on s
\readline{}\reply % read the answer
```

(See the description of `\readline` in `dialog.doc`
cial characters have category 12 except for brace
mended placement of the braces: no closing brac
the very last one. Because of the special catco
nal three arguments, a `^^M` or `%` between argu
character or category-12 character respectively, i
ally, after some rather difficult programming, I m
just about anything (except brace characters) b
be ignored, so the recommended style is not ma
line of each argument are stripped off anyway in
connections with `\menuprefix` etc.; see below.

Menu functions created by `\fmenu` are allowed
functions created with `\fmesj` (from `dialog.sty`
serted at the time of use. This makes it possible
the same menu function if there are only minor

```
\menuprefix, \menusuffix
\inmenuA, \inmenuB
```

The text `\menuprefix` will be added at the begi
will be added at the end. The text `\inmenuA` an
the first and second, respectively second and t
fault values produce a blank line on screen. (Bu

first part is empty, and \inmenuB will be omitted if the last part is empty.) To change any of these texts, use \storemesj or \storexmesj. For example:

```
\storemesj\menuprefix{********* MENU **********}
```

\menuprompt

Furthermore, the function \menuprompt is called at the very end of the menu, so that for example a standard prompt such as Enter a number: could be applied at the end of all menus, if desired. To change \menuprompt, use \fmesj or \fxmesj.

\menuline, \endmenuline
\menutopline, \menubotline

Each line in the middle argument of \fmenu (the list of choices) is embedded in a statement \menuline...\endmenuline. The default definition of \menuline is to add two spaces at the beginning and a newline at the end. Lines in the top or bottom part of the menu are embedded in \menutopline...\endmenuline or \menubotline...\endmenuline respectively. (Notice that all three share the same ending delimiter; if different actions are wanted at the end of a top or bottom line as opposed to a middle menu line, they must be obtained by defining \menutopline or \menubotline to read the entire line as an argument and perform the desired processing.)

An enclosing box for a menu can be obtained by defining \menuline and its relatives appropriately and using \fxmenu (see below).

```
\fxmenu\foobar{
 ⟨preliminary text⟩
}{
 ⟨menu lines⟩
}{
 ⟨following text⟩
}
```

Similar to \fmenu but with full expansion in each part of the text, as with \xmesj.

To get an enclosing box for a menu, write \. at the end of each menu line (to protect the preceding spaces from TEX's propensity to remove character 32 at the end of a line, regardless of its catcode), and then make sure that \menuline and \endmenuline put in the appropriate box-drawing characters on either side. I.e.:

```
\fxmenu\foobar{
First line
Second line
}{
Third line
...
}{
Last line
}
```

With the /o option of emTEX, you can use the b
dard PC DOS character set.

```
\nmenu\Alph\foobar#1{
 ⟨preliminary text⟩
}{
 ⟨menu lines⟩
}{
 ⟨following text⟩
}
```

\nmenu and \nxmenu are like \fmenu, \fxmenu ex
ber each line of the middle part of the menu. (Th
or deleted without tedious renumbering.) The fi
numbers to be used: \alph, \Alph, \arabic, \
These are not yet implemented.

The function \menunumber (taking one argu
cally generated number. The default value is to

```
\def\menunumber#1{[#1] }
```

but by redefining \menunumber you can add pa
have you around each number. Internally a line c
as

```
\menuline\menunumber{5}Text text ...\enc
```

\optionexec\answer

This is a companion function for \readChar and
see if the answer is equal to any one of the ch
cutes \moption? or \moptionQ or \moptionX res

```
\csname moption\curmenu C\endcsname
```

where C means the character that was read and
the current location in the menu system. (\opti

when going between menus, to keep it up to date.) If this control sequence is undefined, \optionexec gives a generic "Sorry, I don't understand" message and repeats the current menu.

Thus the major work involved in making a menu system is to define the menu screens using \fmenu, \fxmenu, and then define corresponding functions \moptionXXX that display one of the menu screens, read a menu choice, and call \optionexec to branch to the next action.

```
\specialhelp\answer{Substitute message}
```

As it turns out, it is sometimes desirable to substitute some other message in place of the generic "Sorry, I don't understand" message given by \optionexec. For instance, suppose a given menu choice leads to a secondary prompt where you ask the user to enter a number of columns for printing some data. If the user accidentally mistypes 0, it would be better to respond with something like

```
Number of columns must be greater than 0.
```

than with the generic message. The function \specialhelp allows you to do this. The first argument is the name of the macro that will be passed to \optionexec. \specialhelp modifies that macro to a flag value that will trigger the substitute message. (But does it carefully, so that you can still use the macro naturally in the substitute message text.)

```
\optionfileexec\answer
```

Like \optionexec, but gets the next menu from a file instead of from main memory, if applicable. This is not yet implemented. The technical complications involved in managing the menu files are many—for example: How do you prevent the usual file name message of TEX from intruding on your carefully designed menu screens, if \input is used to read the next menu file? Alternatively if you try to use \read to read the next menu file, how do you deal with catcode changes?

```
\lettermenu{MN}
```

This is an abbreviation for

```
\menuMN \readChar{Q}\reply \optionexec\reply
```

It calls the menu function associated with the menu name MN, reads a single uppercase letter into \reply, and then calls \optionexec to branch to the case selected by the reply.

```
\if\xoptiontest\answer ... \else ... \fi
```

The function \xoptiontest is for use with \readline or \xreadline, to trap the special responses ? Q q X x before executing some conditional code. It returns a 'true' value suitable for \if testing, if and only if the replacement text of \answer is a single character matching one of those listed. T
ing for a response that can be an arbitrary string o
the user still to get help or quit with the same on
ognized in other situations.

2.2 Implementation

Standard package identification:

```
%<*2e>
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{menus}[1994/11/08 v0.9x]
```

Load the dialog package if necessary.

```
\RequirePackage{dialog}
%</2e>
```

This file requires grabhedr.sty and dialog
ready loaded, load it now and call \fileversio
\inputfwh to *this* file. See the documentation o

```
%<*209>
\csname trap.input\endcsname
\input grabhedr.sty \relax
\fileversiondate{menus.sty}{0.9x}{1994/1
\inputfwh{dialog.sty}
%</209>
```

2.3 Definitions

We start by using the \localcatcodes functio
rent catcodes and set new catcodes for certain s
at more length in dialog.doc.

```
\localcatcodes{\@{11}%
  \~{13}\"{12}\#{6}\^{7}\‘{12}\${3}\:{12}
```

\menuprefix is a string added at the beginning
tle (or uglify it a little, depending on your taste).
70 characters, not counting the two newline cha
get embedded newlines corresponding to the one
extra line break (where the newline character is
this fit in the current column width.]

```
\storexmesj\menuprefix{
==================================%
==================================
}
```

The default value for \menusuffix is the same as for \menuprefix.

```
\let\menusuffix=\menuprefix
```

The default for \inmenuA and \inmenuB is a single newline, which will produce a blank line on screen because they will occur after an \endmenuline, which also contains a newline.

```
\storemesj\inmenuA{
}
\storemesj\inmenuB{
}
```

The default value for \menuline is two spaces. This means that each line in the middle section of a menu defined by \fmenu or \fxmenu will be indented two spaces.

```
\storemesj\menuline{  }
```

By default, no spaces are added at the beginning of a line in the top or bottom section of a menu:

```
\def\menutopline{}
\def\menubotline{}
```

\endmenuline is just a newline.

```
\storemesj\endmenuline{
}%
```

This definition of \menunumber adds square brackets and a following space around each item number.

```
\def\menunumber#1{[#1] }
```

This definition of \menuprompt is suitable for the purposes of listout.tex but will probably need to be no-op'd or changed for other applications.

```
\def\menuprompt{\promptmesj{Your choice? }}
```

Each of the three pieces of a menu gets its own token register.

```
\newtoks\menufirstpart
\newtoks\menuchoices
\newtoks\menulastpart
```

The 'arguments' of \fmenu are #1 menu name, #
preliminary text, #4 list of menu choices, #5 fo
only the first two because we want to change som
ers. The auxiliary function \fxmenub is shared w

Because of the catcode changes done by \me
cent signs between the three final arguments wi
this, we use the peculiar #{ feature of TEX, in in
\@tempa, to read and discard anything that ma
and the next opening brace. Token register assig
ments proper.

```
\def\fmenu#1#2#{\mesjsetup
  \catcode`\#=6 % for parameters
  \toks@{\fxmenub{\gdef}{\begingroup}{}#
  \def\@tempa##1##{%
    \def\@tempa####1####{%
      \def\@tempa{\the\toks@}%
      \afterassignment\@tempa \menulastpa
    \afterassignment\@tempa \menuchoices
  \afterassignment\@tempa \menufirstpart
}
```

Before proceeding to define \fxmenub, we mus
will have to work with is three pieces of text in th
\menuchoices, and \menulastpart, containing
breaks, including possibly but not necessarily ^^
of each piece. What we would like to do, for each
there is one, and the last one, if there is one. Th
this.

If you are one of those rare TEX hackers wi
stand the workings of \stripcontrolMs, the be
action with \tracingmacros = \tracingcomman
low my labored commentary below.

```
\begingroup % localize \lccode change
\lccode`\~=`\^^M
```

The functions \stripM, \stripMa, \stripMb,
\stripcontrolMs. They all carry along an extra
ken register originally given to \stripcontroll
\stripMd we can carry out the assignment of th

When \stripM is called, it should be called l

```
% \expandafter\stripM\expandafter$\the\s
% $^^M$$\stripM\sometoks
%
```

That is, $ should be added at the beginning and $^^M$$ at the end of the text to be processed. And \expandafter's should be added to pre-expand the token register.

These examples illustrate the possible contents of (e.g.) \menufirstpart, before processing

(a) ^^Maaa^^Mbbb^^M

(b) aaa^^Mbbb

(c) ^^Maaa^^Mbbb

(d) aaa^^Mbbb^^M

The processing of example (a) would proceed as follows. Call \stripM, adding $ at the beginning and $^^M$$ at the end.

```
% \stripM $^^Maaa^^Mbbb^^M$^^M$$\stripM
%
```

This finds a match with the $^^M at the beginning. The remaining text is passed on to \stripMb. We know that there is now an extra $^^M$$ at the end, but we don't know if the first $ is preceded by ^^M.

```
% \stripMb aaa^^Mbbb^^M$^^M$$\stripMb
%
```

If it turns out that #2 = $, then there was *not* a ^^M at the end of the original text, and we need to strip off a last remaining $ sign. Otherwise we are finished if we just discard #2 and #3 (the remaining ^^M and $ characters of the ones that we added).

We use $ as a marker since any $ characters that happen to occur in the menu text will have category 12 and thus not match the category-3 $ used in the definition of \stripMa etc. A control sequence could also be used as a marker if we took care to give it a bizarre name that would never arise in the menu text (\fxmenub is used not just by \fmenu but also by \fxmenu, which might have arbitrary control sequences in its arguments); but that means using up one more hash table entry and additional string pool.

```
\lowercase{%
\long\gdef\stripM#1$~#2#3\stripM#4{%
  \ifx$#2%
    \stripMa#1\stripMa#4%
  \else
    \stripMb#2#3\stripMb#4%
  \fi
}
}% end lowercase


\lowercase{%
\long\gdef\stripMa $#1\stripMa#2{%
  \stripMb#1$~$$\stripMb#2}
}% end lowercase


\lowercase{%
\long\gdef\stripMb#1~$#2#3\stripMb#4{%
  \ifx$#2%
    \stripMc#1\stripMc#4%
  \else
    \stripMd#1\stripMd#4%
  \fi
}
}% end lowercase


\long\gdef\stripMc#1$#2\stripMc#3{%
  \stripMd#1\stripMd#3}


\long\gdef\stripMd#1\stripMd#2{#2{#1}}
\endgroup
```

Some tests.

```
% %\lowercase{\def\test#1{\stripM $#1$~$$
% %\tracingmacros=2 \tracingcommands=2 \t
% %\test{~aaa~bbb~}
% %\test{aaa~bbb}
% %\test{~aaa~bbb}
% %\test{aaa~bbb~}
% %\tracingmacros=0 \tracingcommands=0 \t
% %}% end lowercase
%
```

The argument of \stripcontrolMs is a token re
ter will be stripped of a leading and trailing ^^M
the remainder text will be left in the token regis

```
\begingroup \lccode`\~=`\^^M
\lowercase{%
\gdef\stripcontrolMs#1{\xp@\stripM
  \xp@$\the#1$~$$\stripM#1}
}% end lowercase
```

```
\lowercase{%
\gdef\addmenulines#1#2#3{%
```

Add #2 at the beginning and #3 at the end of every line of token register #1.

```
  \def ~##1~##2{%
    #1\xp@{\the#1#2##1#3}%
    \ifx\end##2\xp@\@gobbletwo\fi~##2}%
  \edef\@tempa{\nx@~\the#1\nx@~}#1{}%
  \@tempa\end}
}% end lowercase
\endgroup % restore lccode of ~
```

The function \fxmenub is the one that does most of the hard work for \fmenu and \fxmenu. Argument #4 is the name of the menu, #5 is the argument specifiers (maybe empty). Arguments #1#2#3 are assignment type, extra setup, and expansion control; specifically, these arguments are \gdef \begingroup \empty for \fmenu or \xdef \xmesjsetup and an extra \noexpand for \fxmenu.

That this function actually works should probably be regarded as a miracle rather than a result of my programming efforts.
[3]

```
\def\fxmenub#1#2#3#4#5{%
  \stripcontrolMs\menufirstpart
  \stripcontrolMs\menulastpart
  \stripcontrolMs\menuchoices
  \addmenulines\menuchoices\menuline\endmenuline
  \actively\let\^^M\relax % needed for \xdef
```

Define #4. Expansion control is rather tricky because of the possibility of parameter markers inside \menufirstpart, \menuchoices or \menulastpart.

```
  \toks@{\long#1#4#5}% e.g. \xdef\foo##1##2
```

If \menufirstpart is empty, we don't add the separator material \inmenuA.

```
  \edef\@tempa{\the\menufirstpart}%
  \ifx\@tempa\@empty
    \let\nxa@\@gobble
  \else
    \addmenulines\menufirstpart
      \menutopline\endmenuline
    \let\nxa@\nx@
  \fi
```

If \menulastpart is empty, we don't add the sep

```
 \edef\@tempa{\the\menulastpart}%
  \ifx\@tempa\@empty
    \let\nxb@\@gobble
  \else
    \addmenulines\menulastpart
      \menubotline\endmenuline
    \let\nxb@\nx@
  \fi
```

Set up the definition statement that will create t or \xmesjsetup.

```
\edef\@tempa{{#3\nx@#3#2%
    \def#3\nx@#3\mesjtext{%
      #3\nx@#3\menuprefix
      \the\menufirstpart #3\nxa@#3\inmenu
      \the\menuchoices #3\nxb@#3\inmenuB
      \the\menulastpart #3\nx@#3\menusufi
    #3\nx@#3\sendmesj
    #3\nx@#3\menuprompt}}%
  \toks2 \xp@{\@tempa}%
  \edef\@tempa{\the\toks@\the\toks2 }%
```

Temporarily \relaxify \menuline etc. in order sion if \xdef is applied.

```
  \let\menutopline\relax \let\menuline\re
  \let\menubotline\relax \let\endmenuline
  \let\menunumber\relax
  \@tempa % finally, execute the \gdef or
  \endgroup % matches \mesjsetup done by
}% end \fxmenub
```

Expanding analog of \fmenu.

```
\def\fxmenu#1#2#{\xmesjsetup
  \toks@{\fxmenub{\xdef}{\xmesjsetup}\nx@
  \def\@tempa##1##{%
    \def\@tempa####1####{%
```

[3] Let's see, three miracles is a prerequisite for sainthood in the Catholic church—only two more needed for Don Knuth to be a candidate ...

```
      \def\@tempa{\the\toks@}%
      \afterassignment\@tempa \menulastpart}%
    \afterassignment\@tempa \menuchoices}%
  \afterassignment\@tempa \menufirstpart
}


\def\notyet#1{%
  \errmessage{Not yet implemented:  \string#1}}
```

These two functions aren't implemented yet.

```
\long\def\nmenu#1#2#3#4#5{\notyet\nmenu}
\long\def\nxmenu#1#2#3#4#5{\notyet\nxmenu}
```

2.4 Menu traversal functions

For reliable travel up and down the menu tree, we need to push and pop the value of `\curmenu` as we go along. Among other things, `\curmenu` is used to repeat the current menu after a help message.

```
\newtoks\optionstack


\let\curmenu\@empty
```

Start of a stack element.

```
\let\estart\relax
```

End of a stack element.

```
\let\eend\relax


\def\pushoptions#1{%
  \edef\pushtemp{\estart
    \def\nx@\curmenu{\curmenu}%
    \eend
    \the\optionstack}%
  \global\optionstack\xp@{\pushtemp}%
  \edef\curmenu{\curmenu#1}%
}
```

```
\def\popoptions{%
  \edef\@tempa{\the\optionstack}%
  \ifx\@empty\@tempa
    \errmessage{Can't pop empty stack
      (\string\optionstack)}%
  \else
    \def\estart##1\eend##2\@nil{%
      \global\optionstack{##2}%
      \let\estart\relax##1}%
    \the\optionstack\@nil
  \fi
}
```

The `X` option is a total exit from the menu maze, returns you to the previous menu level.

```
\fmesj\moptionX{Exiting . . .}


\def\repeatoption{%
  \csname moption\curmenu\endcsname}


\def\moptionQ{\popoptions \repeatoption}
```

The sole reason for using `\fxmesj` rather than ment out the initial newline, as the line break wa ing of this documentation within a narrow colum

```
\fxmesj\badoptionmesj#1{%
?---I don't understand "#1".}
```

The function `\optionexec` takes one argume `\curmenu` to determine the next action. The ar containing a single letter, the most recent menu

Common options such as `?`, `Q`, or `X` that may tem are handled specially, to cut down on the n needed for a csname implementation of the menu

```
\def\optionexec#1{%
  \if ?#1\relax \let\@tempa\moptionhelp
  \else \if Q#1\relax
    \ifx\curmenu\@empty \let\@tempa\mopti
    \else \let\@tempa\moptionQ \fi
  \else \if X#1\relax \let\@tempa\moptio
  \else
```

Because special characters, including backslash, are deactivated by \readChar, we can apply \csname without fearing problems from responses such as \relax.

```
    \xp@\let\xp@\@tempa
      \csname moption\curmenu#1\endcsname
    \ifx\@tempa\relax
      \badoptionmesj{#1}\let\@tempa\repeatoption
    \else
      \pushoptions{#1}%
    \fi
  \fi\fi\fi
```

We save up the next action in \@tempa and execute it last, to get tail recursion.

```
  \@tempa
}
```

Really big menu systems could get around TEX memory limits by storing individual menus or groups of menus in separate files and using \optionfileexec in place of \optionexec to retrieve the menu text from disk storage instead of from main memory. However there are a number of technical complications and I probably won't get around to working on them in the near future.

```
\def\optionfileexec#1{\notyet\optionfileexec}
```

The function \xoptiontest must return true if and only if the macro #1 consists entirely of one of the one-letter responses ? Q q X x that correspond to special menu actions. The rather cautious implementation with \aftergroup avoids rescanning the contents of #1, just in case it contains anything that's \outer.

```
\def\xoptiontest#1{TT\fi
  \begingroup \def\0{?}\def\1{Q}%
  \def\2{q}\def\3{x}\def\4{X}%
    \aftergroup\if\aftergroup T%
    \ifx\0#1\aftergroup T%
    \else\ifx\1#1\aftergroup T%
    \else\ifx\2#1\aftergroup T%
    \else\ifx\3#1\aftergroup T%
    \else\ifx\4#1\aftergroup T%
    \else \aftergroup F%
    \fi\fi\fi\fi\fi
  \endgroup
}
```

Default help message, can be redefined if necessary. The extra newlines commented out with % are here only for convenient printing within a narrow column width.

```
\fxmesj\menuhelpmesj{&\menuprefix%
A response of Q will usually send you bac
the previous menu.
A response of X will get you entirely out
the menu system.
&\menusuffix%
Press the <Return> key ( Enter ) to conti
}


\def\moptionhelp{%
  \menuhelpmesj \readline{}\reply \repeat
```

\moptionhelp is the branch that will be taken if response to a menu.

```
\def\moptionhelp{%
  \menuhelpmesj \readline{}\reply \repeat


\xp@\def\csname moption?\endcsname{%
  \moptionhelp}
```

The function \specialhelp can be used to prov sage tailored to a specifc response given by the (the macro containing the response) to contain ? use the message text given in arg #2.

```
\def\specialhelp#1#2{%
  \let\specialhelpreply=#1\def#1{?}\begi
  \def\menuhelpmesj{\let#1\specialhelpre
    \promptxmesj{#2\
Press <return> to continue:}\endgroup}%
}
```

Init.

```
\def\specialhelpreply{}
```

This is a convenient abbreviation for an often-us

```
\def\lettermenu#1{%
  \csname menu#1\endcsname
  \readChar{Q}\reply \optionexec\reply
}
```

Restore any catcodes changed locally, and de

```
\restorecatcodes
\endinput
```

Appendix
Miscellaneous support functions: grabhedr.sty

A.1 Introduction

This file defines a function \inputfwh to be used instead of \input, to allow TEX to grab information from standardized file headers in the form proposed by Nelson Beebe during his term as president of the TEX Users Group. Usage:

```
\inputfwh{file.nam}
```

Functions \localcatcodes and \restorecatcodes for managing catcode changes are also defined herein, as well as a handful of utility functions, many of them borrowed from latex.tex: \@empty, \@gobble, \@gobbletwo, \@car, \@@input, \toks@, \afterfi, \fileversiondate, \trap.input.

The use of \inputfwh, \fileversiondate, and \trap.input as illustrated in dialog.sty is cumbersome klugery that in fact would better be handled by appropriate functionality built into the format file. But none of the major formats have anything along these lines yet. (It would also help if TEX made the current input file name accessible, like \inputlineno.)

A.2 Implementation

Standard package identification:

```
%<*2e>
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{grabhedr}[1994/11/08 v0.9j]
%</2e>
```

By enclosing this entire file in a group, saving and restoring catcodes 'by hand' is rendered unnecessary. This is perhaps the best way to locally change catcodes, better than the \localcatcodes function defined below. But it tends to be inconvenient for the TEX programmer: every time you add something you have to remember to make it global; if you're like me, you end up making every change twice, with an abortive test run of TEX in between, in which you discover that a certain control sequence is undefined because you didn't assign it globally. (Using \globaldefs = 1 is dangerous in my experience; you have to take care not to accidentally change any variables that you don't want to be changed globally.)

```
\begingroup
```

Inside this group, enforce normal catcodes. All definitions must be global in order to persist beyond the \endgroup.

```
\catcode96 12 % left quote
\catcode`\= 12
\catcode`\{=1 \catcode`\}=2 \catcode`\#=6
\catcode`\$=3 \catcode`\~=13 \catcode`\^=
\catcode`\_=8 \catcode`\^^M=5 \catcode`\%
```

Make @ a letter for use in 'private' control sequ

```
\catcode`\@=11
```

A.3 Preliminaries

For \@empty, \@gobble, ... we use the LATEX used with LATEX we won't waste hash table and

Empty macro, for \ifx tests or initialization of

```
\gdef\@empty{}
```

Functions for gobbling unwanted tokens.

```
\long\gdef\@gobble#1{}
\long\gdef\@gobbletwo#1#2{}
\long\gdef\@gobblethree#1#2#3{}
```

The function \@car, though not really needed b principal customers of grabhedr.sty (e.g., dial

```
\long\gdef\@car#1#2\@nil{#1}
```

To define \@@input as in LATEX we want to let But if a LATEX format is being used we don't wa cause by now \input has changed its meaning. used it behooves us to check, before defining \@ its primitive meaning. Otherwise there's a good properly.

```
\ifx\UndEFiNed\@@input % LaTeX not loaded
```

This code shows a fairly easy way to check wheth trol sequence is still the original meaning.

```
  \edef\0{\meaning\input}\edef\1{\string\
  \ifx\0\1%
    \global\let\@@input\input
  \else
    \errhelp{%
Grabhedr.sty needs to know the name of th
\input primitive in order to define \inpu
properly. Consult a TeXnician for help.}
    \errmessage{%
```

```
      Non-primitive \noexpand\input detected}%
  \fi
\fi
```

Scratch token register.

```
\global\toksdef\toks@=0
```

Sonja Maus's function for throwing code over the `\fi` ("An Expansion Power Lemma", *TUGboat* vol 12 no 2 June 1991). (Except that she called this function `\beforefi`.)

```
\long\gdef\afterfi#1\fi{\fi#1}
```

We will be using `\noexpand` a lot; this abbreviation improves the readability of the code.

```
\global\let\nx@\noexpand
```

Another convenient abbreviation.

```
\global\let\xp@\expandafter
```

A.4 Reading standard file headers

The function `\inputfwh` ('input file with header') inputs the given file, checking first to see if it starts with a standardized file header; if so, the filename, version and date are scanned for and stored in a control sequence.

For maximum robustness, we strive to rely on the fewest possible assumptions about what the file that is about to be input might contain.

Assumption 1: Percent character `%` has category 14. I.e., if the first line of the file to be input starts with `%`, it is OK to throw away that line.

```
\begingroup \lccode`\.=`\%%
\lowercase{\gdef\@percentchar{.}}%
\endgroup
```

The function `\fileversiondate` is not only a useful support function for `\inputfwh`, it can also be used by itself at the beginning of a file to set file name, version, and date correctly even if the file is input by some means other than `\inputfwh`—assuming that the arguments of the `\fileversiondate` command are kept properly up to date.

```
\gdef\fileversiondate#1#2#3{%
  \xp@\xdef\csname#1\endcsname{#2 (#3)}%
  \def\filename{#1}\def\fileversion{#2}%
  \def\filedate{#3}%
  \message{#1 \csname#1\endcsname}%
}
```

And now apply `\fileversiondate` to this file.

```
%<*209>
\fileversiondate{grabhedr.sty}{0.9j}{1994
%</209>
```

`filehdr.el` by default adds a string of equal si
fix) at the very top of a file header. This string
we can start looking for the real information of t

```
\xdef\@filehdrstart{%
  \@percentchar\@percentchar\@percentchar
  ==================================%
  ==================================}
```

The purpose of this function is just to scan up to
beginning of the file header body. Everything be
our present purposes.

```
\gdef\@scanfileheader#1@#2#{\@xscanfileh
```

Throw in some dummy values of version and d
quire from a file header is that the filename field

```
\long\gdef\@xscanfileheader#1{%
  \@yscanfileheader#1{} version = "??",
  date = "??",\@yscanfileheader}
```

This function assumes that filename, version, and
der (but not necessarily adjacent). It's possibl
missing, or out of order, but the corresponding
`\filedate` will not get set properly unless the
[...,] date. Trying to handle different orderings
yet been struck by a suitable flash of insight on h
consuming picking apart of the entire file header

```
\long\gdef\@yscanfileheader
  #1 filename = "#2",#3 version = "#4",%
  #5 date = "#6",#7\@yscanfileheader{%
  \endgroup
  \csname fileversiondate\endcsname{#2}{#
}
```

This function has to look at the first line of the file to see if it has the expected form for the first line of a file header.

```
\begingroup
\lccode`\$=`\^^M
\lowercase{\gdef\@readfirstheaderline#1$}{%
  \toks@{#1}%
  \edef\@tempa{\@percentchar\the\toks@}%
  \ifx\@tempa\@filehdrstart
    \endgroup \begingroup
    \catcode`\%=9 \catcode`\^^M=5 \catcode`\@=11
```

Double quote and equals sign need to be category 12 in order for the parameter matching of `\@xscanfileheader` to work, and space needs its normal catcode of 10.

```
    \catcode`\ =10 \catcode`\==12 \catcode`\"=12
  \xp@\@scanfileheader
  \else
    \message{(* Missing file header? *)}%
    \afterfi\endgroup
  \fi}
\endgroup
```

An auxiliary function.

```
\gdef\@xinputfwh{%
  \ifx\next\@readfirstheaderline
```

Sanitize a few characters. Otherwise an unmatched brace or other special character might cause a problem in the process of reading the first line as a macro argument.

```
    \catcode`\%=12 \catcode`\{=12 \catcode`\}=12
    \catcode`\\=12 \catcode`\^^L=12
    \catcode`\^=12
%    Unique terminator token for the first line.
    \catcode`\^^M=3\relax
  \else \endgroup\fi
}
```

Auxiliary function, carries out the necessary `\futurelet`.

```
\gdef\@inputfwh{\futurelet\next\@xinputfwh}
```

Strategy for (almost) bulletproof reading of the first line of the input file is like this: Give the percent sign a special catcode, then use `\futurelet` to freeze the catcode of the first token in the input file. If the first token is *not* a percent character, then fine, just close the group wherein the percent character had its special catcode, and proceed with normal input; the fi code because we did not change anything except we still proceed with 'normal' input execution, ing it suitably, we can carry out further tests expected form (three percent signs plus lots of e

```
\gdef\inputfwh#1{%
  \begingroup\catcode`\%=\active
  \endlinechar`\^^M\relax
  \lccode`\~=`\%\relax
  \lowercase{\let~}\@readfirstheaderline
  \xp@\@inputfwh\@@input #1\relax
}
```

A.5 Managing catcode changes

A survey of other methods for saving and restor than I have time for at the moment. The me know (other methods use up one extra control s bustly handle multiple levels of file nesting).

The `\localcatcodes` function changes catcodes pairs given in its argument, saving the previou ters on a stack so that they can be retrieved late ple:

```
\localcatcodes{\@{11}\"\active}
```

to change the catcode of `\@` to 11 (letter) and In PLAIN TEX you'd better be careful to use `+` `\localcatcodes` because of the outerness of `\+`.

This function works by using token registers signment statements: in `\toks0` we put the sta codes to their previous values, while in `\toks 4` set catcodes to their new values.

```
\gdef\localcatcodes#1{%
  \ifx\@empty\@catcodestack
    \gdef\@catcodestack{{}}%
  \fi
  \def\do##1##2{%
    \ifnum##2>\z@
      \catcode\number`##1 \space
      \number\catcode`##1\relax
    \expandafter\do\fi}%
```

```
  \xdef\@catcodestack{{\do#1\relax\m@ne}%
    \@catcodestack}%
  \def\do##1##2{\catcode`##1 ##2\relax\do}%
  \do#1\ {\catcode32\let\do}%
}
```

Init the stack with an empty element; otherwise popping the next-to-last element would wrongly remove braces from the last element. But as a matter of fact we could just as well initialize `\@catcodestack` to empty because `\localcatcodes` is careful to add an empty final element if necessary.

```
\gdef\@catcodestack{{}}
```

The function `\restorecatcodes` has to pop the stack and execute the popped code.

```
\gdef\restorecatcodes{%
  \begingroup
  \ifx\@empty\@catcodestack
    \errmessage{Can't pop catcodes;
      \nx@\@catcodestack = empty}%
    \endgroup
  \else
    \def\do##1##2\do{%
      \gdef\@catcodestack{##2}%
```

Notice the placement of `#1` after the `\endgroup`, so that the catcode assignments are local assignments.

```
      \endgroup##1}%
    \xp@\do\@catcodestack\do
  \fi
}
```

A.6 Trapping redundant input statements

The utility `listout.tex` calls `menus.sty`, which calls `dialog.sty`, and all three of these files start by loading `grabhedr.sty` in order to take advantage of its functions `\fileversiondate`, `\localcatcodes`, and `\inputfwh`. But consequently, when `listout.tex` is used there will be two redundant attempts to load `grabhedr.sty`. The straightforward way to avoid the redundant input attempts would be to surround them with an `\ifx` test:

```
\ifx\undefined\fileversiondate
  \input grabhedr.sty \relax
  \fileversiondate{foo.bar}{0.9e}{10-Jun-1993}
\fi
```

This method has a few drawbacks, howeve
open throughout the processing of everyth
`\fileversiondate` statement, which makes any
harder to debug; (2) if `\undefined` becomes acci
fail; (3) choosing the right control sequence to test
tle care.

In a situation where we know that the file to b
applied to it, if it was already input, then we ha
we can test to find out whether the file has alre
file. Assuming a standard form for the input stat
ther plain TEX or LATEX, and makes as few assu
a function that will trap input statements and
has not yet been loaded:

```
\csname trap.input\endcsname
\input grabhedr.sty \relax
\fileversiondate{foo.bar}{1.2}{1993-Jun-
```

The function `\trap.input` scans for an input st
ecutes it if and only if the file has not yet been
trol sequence consisting of the file name is undef
had `\fileversiondate` applied to it). The can
the best is `\input` ⟨*full file name*⟩␣`\relax`. Hav
put statement will not try to expand beyond the
catcoded to 9 (ignore), as is done rather frequently
mers. The `\relax` would ordinarily render the sp
but I prefer leaving the space in to avoid interfer
take a space-delimited argument that are occas
cial effects (see, for example, "Organizing a large
Binding, *Cahiers GUTenberg*, numéro 10–11, sept
ment form `\input{...}` cannot, unfortunately, b
compatibility is required.

```
\expandafter\gdef\csname trap.input\endcs
  \input#1 \relax{%
    \expandafter\ifx\csname#1\endcsname\r
      \afterfi\inputfwh{#1}\relax
    \fi}
```

End the group that encloses this entire file, a

```
\endgroup
\endinput
```