are available on CD-Rom, or user groups that can be of assistance in obtaining materials.

Finally, it is customary to mention briefly the layout and design of any book about TEX. Although the author and publisher consider this book an 'excellent example' of a beautiful book typeset with TEX, I do not entirely agree. With the many white spaces around illustrations and headings, it is a less than felicitous idea to separate paragraphs by white space, instead of simply indenting them. Also, the only distinction between levels of headings is through font sizes. Any real size difference, however, is dwarfed by the difference between headings in all lowercase such as **xv**, and ones in the same font size but mostly uppercase such as **PBMPlus**.

In conclusion I can safely state that this book stands alone among TEX books. It contains much that is not in print anywhere else, and its usefulness spans all types of TEX users. Victor sez check it out.

◇ Victor Eijkhout
Department of Computer Science
University of Tennessee at
Knoxville
Knoxville TN 37996-1301
Internet: eijkhout@cs.utk.edu

However... I disagree with the publishers: I do not understand why they have published this book. I mean, why so late, or why they have not waited for Goossens and Mittelbach's *LATEX Companion* translation? This book doesn't have exactly the same audience (Rolland's is more directed to beginners, not low level ones but rather, let us say, regular users, while the *Companion* is instead for advanced LATEXers). No, the problem is that Rolland's book is rather old: nothing about multicolumns, nothing (or almost) about NFSS, nothing about the use of PostScript files (through psfig, pstricks, etc.). Not one Frequently Asked Question is answered here. The author is a beginner and he wrote the book he would have liked to have when learning LATEX. On the other hand, the publisher should have asked (for example to the French TUG: Association GUTenberg) for opinions on this book. It would have been said that chapters should have been added (LATEX in 1994 is not the same as in 1986...). At least, let us hope that a French translation of *The Companion* will be published as soon as possible.

◇ Jacques André
IRISA University of Rennes
Campus de Beaulieu
F-35042 Rennes Cedex, France
Jacques.Andre@irisa.fr

Book review: *LATEX guide pratique*

Jacques André

Christian ROLLAND, *LATEX guide pratique*. Éditions Addison-Wesley-France, Paris, 1993, 280 pages; ISBN 2-87908-025-8; in French.

Up to now, only two books have been available in French on TEX or LATEX: the famous *Petit livre de TEX* by Seroul, and a translation, by Éric Cornelis, of Michael Urban's *An introduction to LATEX* published as *Premier pas en LATEX* by *Cahiers GUTenberg*. The gap between these two books is now filled.

Although it follows more or less the same road as Lamport's *LATEX user's guide*, this French guide is not just a translation of it. It contains a lot of useful macros, examples, tables, index, etc. Furthermore, this book contains details on extensions that are not described in Lamport's book (such as on Makeindex, BIBTEX, Picture, and, an important point, `french.sty`, etc.).

I used to have Lamport's book near my station. Now, as a French speaker, I have Rolland's: this is a good book!

Typesetting on Personal Computers

NextTEX: TEX plus the NextStep Operating System

Alan Hoenig

I recently upgraded my computer system and now use the NextStep operating system on a 486 Intel box. I can't imagine using any other operating system. One of the many pleasures of this computer environment is the implementation of TEX (plus METAFONT plus all other TEX- and METAFONTware) developed by Tom Rokicki of Radical Eye Software for NextStep. The purpose of this article is to describe the unique features of *TEXView*, the name for this system. I will not spend much time on the standard features common to all implementations. As far as I have so far been able to determine,

TEXView contains within it every standard feature you expect to be there.

Since many readers may not be familiar with NextStep, I include some comments at the end of the article. *TEXView* is a standard component of NextStep. NextStep consists of a graphical user interface on top of Unix (BSD 4.3). Windows can be opened in which different tasks may be launched. When you purchase NextStep, you get Unix and the GUI, together with *TEXView* and other packages bundled together.

1 NextTEX

It's possible to bring up a window in NextStep which accepts commands in a familiar terminal mode. All of the component programs of *TEXView* may be invoked using familiar command line options. The TEX that is the core of *TEXView* is a "big" TEX which contains two non-standard enhancements. If the first line of a document file is of the form

```
%&foo
```

then the vanilla command `tex myfile` automatically invokes the format file `foo.fmt`. That is, it is equivalent to the command line

```
tex &foo myfile
```

With this convention in place, a single TEX invocation works for plain files, LATEX files, $\mathcal{AMS}$-TEX files, LA$\mathcal{MS}$-TEX files, and so on.

A second enhancement makes it possible to interrupt a TEX run to execute another command. We can do so by simply writing the commands to output stream 18. Example: If the first part of a TEX run prepares a raw index file `idx.r`, we can sort and typeset it in the same pass by issuing commands like

```
\immediate\write18{mysort <idx.r >idx.s}
\input idx.s
```

2 An Integrated System

But the most useful way to use NextTEX is not in the traditional command line mode but as part of the integrated *TEXView* environment. I begin by using the Emacs editor to create `myfile.tex`, say. When finished, I save the file to disk, but I don't exit Emacs. Since Unix is multi-tasking, I can keep Emacs 'up' so it is easy to re-enter it to make the inevitable fixes to my source file. If you prefer, just enter `e` in response to the TEX error prompt, and this TEX turns you and the document over to the system editor. To change the choice of editors, it is necessary only to set an environment variable.

The File Viewer is the main NextStep window which lists in icon form the files I am currently interested in. After making sure that `myfile.tex` is one of these files, I double click on it to launch *TEXView*. (I make sure that the first line of my file is of the form `%&foo`, but purists can ensure that the default version of TEX invokes their favorite format.)

In a moment, a small TEX Command Window opens up; this acts as the console which displays the contents of the `log` file and prompts for corrections when necessary. Just as soon as TEX ships out the first page, the *TEXView* previewer displays this page in a new, large window. It's not necessary to wait for the rest of the job to finish — you can begin scrutiny of your document right away. This preview window is so central to this implementation that it's no wonder Tom calls it *TEXView*.

3 Previewing

NextStep incorporates Display PostScript technology, and this is fully integrated into *TEXView*. So, if your document contains references to outline fonts, or encapsulated PostScript files, they will be fully visible in the preview window. For me, this feature alone is worth the price of admission. (I have been reminded, though, that AmigaTEX has possessed this capability since 1990.)

You can use the mouse to scroll or drag the preview display, and the size of the preview window is itself easily adjusted. A single click of the mouse zooms and unzooms the image, and you have access to a huge range of zoom magnifications as part of options to a Window Command.

A single click on the preview image reports the current position of the mouse. If you click on two different points of the previewed document, *TEXView* will report on the real distance between these points in units either of inches, centimeters, real points, PostScript (big) points, or pico-light-seconds. (I learned from this that a pico-light-second is about 17% larger than a point.) You can improve the accuracy of the click by zooming the preview. I was surprised at how quickly I came to rely on this feature. Whenever a printed element doesn't appear quite where I intended — a frequent occurrence — the double click gives a good idea of the magnitude of the displacement. Having this magnitude in hand often provides a vital clue for correcting the problem, and it's nice to be able to get this hint without printing the document.

TEXView does color. Courtesy of PostScript and of the latest version of `dvips` (which is of course part of the package) it's possible to include color in your document. With a color monitor, you can preview in color. (Otherwise, the colored regions appear in a suitable shade of gray.)

4 Printing the Document

To print the document, I simply click the Print button in the *TEXView* menu. Both *TEXView* (and NextStep, for that matter) expect to print to a PostScript printer. (If your printer is not color, any colored regions appear as a shade of grey on the printed document.) You can also 'print' to fax by pressing a 'fax' button.

5 The Integrated Product

A description of any integrated software system, even when scrupulously accurate, may fail to convey a feel for the success of the implementor in creating an integrated environment which feels right, like an old baseball glove. I don't quite know how such a concept could possibly be quantified, but it is my opinion that *TEXView* succeeds in this endeavor, and succeeds admirably. I have become dependent on *TEXView*'s special features, and my hand has begun to creep naturally toward the mouse at appropriate points in the *TEXView* cycle. It is largely for this reason that I refrain from presenting a table of performance statistics for *TEXView*. What's the point? Normally, it's nice to know speed stats so you know how long you have to wait before you can print or preview. But since *TEXView*'s preview begins long before the TEX compilation is complete, such a consideration is irrelevant. (Subjectively, though, NextTEX seems speedy to me.)

Bringing up NextStep and *TEXView* demands a little more in terms of CPU power and expense than the typical PC user may be used to (see below). Nevertheless, if TEX is a critical part of your computer operations, you might well consider making the switch if only to have access to *TEXView*.

A The NextStep Operating Environment

NextStep is a persnickety operating system. Not just any 486 will do, but only those for whom NextStep has been appropriately tweaked. To find out which computers will work, you will need to call NeXT Computer and ask for their hardware compatibility guide ([800] TRY-NEXT or [800] 848-NEXT).

I am running NextStep on a Logisys computer whose chip is a 486 running at 66 mhz. I have 32 meg of ram and an 820 meg hard disk. NextStep systems don't have to be quite this powerful, but I wanted to indulge myself. In addition, you will need a SCSI CD-ROM drive (the operating system is distributed on a CD-ROM). My system, which includes a Nanao 17 inch SVGA monitor, costs about $5500 all told. At the time of the purchase, the Logisys (with which I am very pleased) was the cheapest NextStep desktop system by quite a bit. It wouldn't surprise me if the situation has changed. In addition to NeXT's hardware compatibility guide, you should check the ads in the journal *NextWorld* (which I can find at the larger newsstands in my home town) for information about competitively priced systems. (Or just wait. Computer power continues to get ever cheaper, and NextStep-able computers will in a year or two surely cost a fraction of their current price.)

In addition to the hardware, you'll need to purchase NextStep. The full package, including developer's version, runs about $1700, which may make DOS users gasp, but appears to be quite competitive with other versions of Unix for the PC. It is possible to get the regular version of NextStep for about $700. If your timing is right and special promotional sales are under way, this might reduce the price further. I purchased NextStep in the fall of 1993, at which time there was a very attractive introductory offer that I was able to take advantage of. I do not know what (if any) special offers might still apply, but you should ask the folks at NeXT when you request the hardware guide.

There are two major FTP sites for NextStep software, although much of this material runs on the now-discontinued Motorola NeXT computers. There are at least half-a-dozen active news groups devoted to NextStep, and the Internet community has proven to be unfailingly courteous and helpful the many times I bugged total strangers while setting up my system.

Any discussion of NextStep, no matter how brief, would be remiss if it did not include some mention of NextStep's graphics. They are stunning. NextStep introduces and demands new æsthetic standards for graphic user interfaces. Graphics aside, the interface itself is far more useful than that of the Macintosh or OS/2, and substantially more so than Microsoft *Windows*. (A third-party *Windows* emulator runs under NextStep so your mountains of *Windows* and DOS software are still usable under NextStep.)

It is too much to hope that this upstart operating system will make much headway against *Windows*. I am having so much fun with it, that I can't help rooting for it anyway.

◇ Alan Hoenig
17 Bay Avenue
Huntington, NY 11743 USA
ajhjj@cunyvm.cuny.edu

Macros

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

Michael Downes

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 PLAINTEX or other common macro packages that include PLAINTEX in their base, such as AMS-TEX or EPLAIN.

The appendix describes `grabhedr.sty`, required by `dialog.sty`, which provides two important file-handling features: (1) 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. And (2) 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 `\reading` the information separately (unreliable due to system-dependent differences in the equivalence of TEX's `\input` search path and `\openin` search path).

These files and a few others (notably `listout.tex`) are combined in a package[1] which should, by the time this article appears, be available on the Internet by anonymous ftp from nodes of CTAN, the Comprehensive TEX Archive Network, e.g., `ftp.shsu.edu` (USA), or `ftp.dante.de` (Europe). The file `listout.tex` is a utility for printing out 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).

[1] Or 'bundle', using more recently established LaTeX terminology for a group of related files, since 'package' now means a LaTeX extension suitable for use with `\usepackage`.

Here's an example from the menu system of `listout.tex` to show how features from `dialog.sty` and `menus.sty` can be put to use. First, the menu that you would see if you wanted to change the font or line spacing:

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

Current settings: typewriter 8 / 10.0pt.

    Q  Quit          X  Exit          ?  Help
================================================

Your choice?
```

Suppose you wanted to change line spacing to 9 points, so you entered `l` and then `9pt`, except that on your first attempt you accidentally mistyped `9pe` instead of `9pt`. Here's what you would see on screen:

```
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 responses, and the value given for line spacing is checked to make sure it's a valid TEX dimension. Before continuing, the internalized version of the user's value is echoed on screen to confirm that the entered value was read correctly.

Now here's how the above menu is programmed in `listout.tex`. A function `\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 calls `\menuF` (given the argument `F`) to print the given menu on screen, 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 in its menus only lowest-common-denominator ordinary printable ASCII characters in the range 32–126. Fancier menus can be obtained at a cost of forgoing system independence, for instance by using emTEX's /o option to output the box-drawing characters in the standard PC DOS character set.

Notation

Double-hat notation such as ^^J is used in this article for control characters, as in *The TEXbook*, although occasionally the alternate form 'CONTROL-J' is used when the emphasis is away from the character's tokenized state inside TEX. A couple of abbreviations from grabhedr.sty are used frequently in the macro code: \xp@ = \expandafter, and \nx@ = \noexpand. Standard abbreviations from plain.tex such as \z@ or \toks@ are used without special comment.

Part 1
Basic dialog functions: dialog.sty

1.1 History

This file, dialog.sty, was born out of a utility called listout.tex that I wrote for my personal use. The purpose of listout.tex was to facilitate printing out plain text files — electronic mail, program source files in various programming languages, and, foremost, TEX macro files and log files. An important part of my TEX programming practice is to print out a macro file on paper and read it through, marking corrections along the way, then use the marked copy as a script for editing the file. (For one thing, this allows me to analyze and mark corrections while riding the bus to work, or sitting out in the back yard to supervise the kids.) The output I normally desired was two 'pages' per sheet of U.S. letter-size paper printed landscape, in order to conserve paper.

Once created, listout.tex quickly became my favorite means of printing out plain text files, not to mention an indispensable tool in my debugging toolbox: I turn on \tracingmacros and \tracingcommands, then print out the resulting log file so that I can see several hundred lines of the log at once (by spreading out two or three pages on my desk with 100+ lines per page); then I trace through, cross things out, label other things, draw arrows, and so forth.

I soon added a filename prompting loop to make it convenient to print multiple files in a single run. In the process of perfecting this simple prompting routine — over two or three years — and adding the ability to optionally specify things like number of columns at run time, eventually I wrote so much dialog-related macro code that it became clear this code should be moved out of listout.tex into its 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 in 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 space after the \ifvmode, \else, and \fi.[2] Since occasionally this may be troublesome, \. is defined inside the argument of \xmesj to be a 'control word terminator': If the expansion of \foo is abc, then \foo\.xyz produces abcxyz on screen (as opposed to \foo xyz which would produce abc xyz). Thus the above conditional could be written as

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

Even though the catcode changes done by \xmesj setup have no effect if \xmesj is used inside an argument or definition replacement text, I find it convenient occasionally to use \xmesj in those contexts, in order to get other aspects of the \xmesj setup. For instance, if you need to embed a message that contains a percent sign inside a definition, you can write

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

To further support such uses of \xmesj, the following changes are also done by \xmesj setup: the backslash-space control symbol \␣ is made equivalent to \space; \^^J and \^^M are defined to produce a \newlinechar; and active tilde ~ will produce a category-12 tilde.

Among other things, this setup makes it easier to obtain newlines and multiple spaces in an embedded message. For example, in the following definition the message will have a line break on screen for each backslash at the end of a line, and the 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 function \barfoo with \f[x]mesj and calling \barfoo inside of \bar would allow more natural entry of the newlines and the multiple spaces, but would be slightly more expensive in string pool and hash table usage.

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

These are like \mesj, \xmesj but use \message rather than \immediate\write16 internally, thus if the following operation is a \read, the user will see the cursor on 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, which means that it cannot actually be printed in the message text. Use of a visible character such as !, rather than the normal \newlinechar ^^J, is

[2] 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.

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, #, 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 macro \answer. (The macro name can be anything chosen by the programmer, not just \answer.) Before reading, all special characters are deactivated, so that the primitive \read will not choke if the user happens to enter something like \newif or CONTROL-L or }. Depending on the operating system, certain characters — e.g., CONTROL-C, CONTROL-Z, CONTROL-D, CONTROL-H — might have special effects instead of being entered into the replacement text of \answer, regardless of the catcode changes. To take the most obvious example, under most operating systems, typing CONTROL-H (the Rubout or Backward-Delete key) will delete the previous character from the user's response, instead of entering an ASCII character 8 into \answer.

There is one significant exception from the catcode changes that are done for \readline: spaces and tabs retain their normal catcode of 10, so that multiple spaces in an answer will be reduced to a single space, and macros with normal space-delimited arguments will work when applied to the answer. (I can't think of any likely scenario where category 12 for spaces would be useful.) Also, the catcode of ^^M is set to 9 (ignore) so that an empty line — meaning that the user just pressed the carriage return/enter key — will result in an empty \answer. If the answer is empty, the given default string will be substituted. The default string can be empty.

\xreadline{⟨*default*⟩}\answer

Like \readline but the answer is read as executable tokens; the usual catcodes of the TEX special characters remain in effect while reading the answer. A few common outer things (\bye, \+, \newif, ^^L, among others) are neutralized before the \read is done, but the user can still cause problems by entering some other outer control sequence or unbalanced braces. I doubt there's any bulletproof solution, if the tokens are to remain executable, short of the usual last resort: reading the answer using \readline, writing it to a file, then inputting the file.

\readchar{⟨*default*⟩}\answer

This is like \readline but it reduces the answer to its first character. ⟨*default*⟩ is either a single character or empty.

\readChar{⟨*default*⟩}\answer

This is like \readchar and also uppercases the answer.

\changecase\uppercase\answer

The function \changecase redefines its second argument, which must be a macro, to contain the same text as before, but uppercased or lowercased according to the first argument. Thus \readChar{Q}\answer is equivalent to

```
\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 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 U.S. 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 paper?
Enter u or U for US letter, %
anything else for A4: }
\readChar{A}\reply % default = A4
\if U\reply \textheight=11in \textwidth=8.5in
\else \textheight=297mm \textwidth=210mm \fi
%    Subtract space for 1-inch margins
\addtolength{\textheight}{-2in}
\addtolength{\textwidth}{-2in}

\promptxmesj{
Left margin setting? %
[Return = keep current value,
\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` functions can be used for this same task, they are rather more awkward, and checking the margin value for validity would be quite difficult.

1.5 Preliminaries

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

```
\csname trap.input\endcsname
\input grabhedr.sty \relax
\fileversiondate{dialog.sty}{0.9v}{6-Jul-1994}%
```

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, `\endlinechar` 5, ␣ 10, % 14, a–z A–Z 11, 0–9 . - 12.)

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

1.6 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.[3]

\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 changes. Then we change all relevant catcodes to 12 except for backslash, open brace, and close brace, which can be handled by judicious application of \escapechar, \string, \edef, and \xdef. By defining \do in a slightly backward way, so that it doesn't take an argument, we don't need to worry about the presence of \+ in the list of control symbols. Notice the absence of \` from the list of control symbols; it was already catcoded to 12 in the \localcatcodes declaration at the beginning of this file — otherwise it would be troublesome to make the definition of \do bulletproof (consider the possibilities 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 \\, \{, and \} to produce the corresponding category-12 character tokens. Setting \escapechar to −1 means that \string will omit the leading backslash that it would otherwise produce when applied to a control sequence.

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

Space and percent are done last. Then, with almost all the special characters now category 12, it's rather easy to define \otherchars.

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

\controlchars is another list for the control characters ASCII 0–31 and 127. The construction of this list is similar to the construction of \otherchars. We need to turn off \endlinechar because the catcode of ^^M is going to be changed. The ^^L inside the \gdef is not a problem (as it might have been, due to the usual outerness of ^^L) because the catcode is changed from 13 to 12 before that point.

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

[3] 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.

```
\endgroup
```

The function \actively makes a given character active and carries out the assignment given as the first argument. The assignment can be embedded inside other definitions 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 \^^M, rather than a single character. (Except that + is safer than \+ in PLAINTEX.) 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 use is in making ^^M active in order to get special action at the end of each line of input. The usual way of going about this would be to write

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

which is a puzzling construction to the TEX novice who doesn't know what \obeylines does with \par. The same effect could be gotten a little more transparently with

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

In the definition of \actively we use the unique properties of \lowercase to create an active character with the right character code, overlapping with a \begingroup \endgroup structure that localizes the necessary lc-code change.

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

The \mesjsetup function starts a group to localize catcode changes. The group will be closed eventually by a separate function that does the actual sending or stores the message text for later retrieval.

We want to change the catcode of each character in the three lists \otherchars, \controlchars, and \highchars to 12. After giving \do a recursive definition, we apply it to each of the three lists, adding a suitable element at the end of the list to make the recursion stop there. This allows leaving out the \do tokens from the character lists, without incurring the cost of an if test at each recursion step.

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

The abbreviation \xp@ = \expandafter is from grabhedr.sty.

```
  \xp@\do\otherchars{a11 \@gobbletwo}%
  \xp@\do\controlchars{a11 \@gobbletwo}%
  \xp@\do\highchars{a11 \@gobbletwo}%
  \actively\edef\^^I{ \space\space\space
    \space\space\space\space}%
```

The convenient treatment of newlines in the argument of \mesj (every line break produces a line break on screen) is achieved by making the ^^M character active and defining it to produce a category-12 ^^J character. Although for \mesj it would have sufficed to make ^^M category 12 and locally set \newlinechar = ^^M 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 under which the message text is to be stored.

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

While \storemesj\foo{...} is more or less the same as \def\foo{...} with special catcode changes, \fmesj\foo{...} corresponds to \def\foo{\mesj{...}}, that is, after \fmesj\foo the function \foo can be executed directly to send the message. Thus \storemesj is typically used for storing pieces of messages, while \fmesj is used for storing entire messages.

To read the parameter text #2, we use the peculiar #{ feature of TEX to read everything up to the opening brace.

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

The parameter text #2 must be stored in a token register rather than a macro to avoid problems with # characters. The \long prefix is just to admit the (unlikely) possibility of using \fmesj to define something such as an error message saying 'You can't use #1 here' where one of the possibilities for #1 is \string\par.

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

Define \@tempa to put together the first two arguments and [pseudo]argument #3 and make the definition of #1.

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

The abbreviation \nx@ = \noexpand is from grabhedr.sty.

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

\xmesjsetup is like \mesjsetup except it prepares to allow control sequence tokens and normal comments in the message text. For TEXnicians' convenience certain other features are thrown in.

Here, unlike the setup for \xreadline, I don't bother to remove the outerness of \bye, \newif, etc., because I presume the arguments of \xmesj, \fxmesj, \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.

```
  \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 no-op, 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, \^^M (i.e., a lone backslash at the end of a line) will produce a newline, also \^^J, while finally \par = blank line translates to two newlines.

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

Define \^^M 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 with expansion.

```
\def\promptxmesj{\xmesjsetup
  \afterassignment\sendprompt \edef\mesjtext}
```

And \storexmesj is like \storemesj, with expansion. Since we allow arguments for the function being defined, we also must define \# to produce a single category-12 # character so that there will be a way to print # in the message text.

```
\def\storexmesj#1#2#{\xmesjsetup
  \catcode`\#=6 % to allow arguments if needed
  \edef\#{\string##}%
  \afterassignment\endgroup
  \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\tw@}%
      \nx@\nx@\nx@\sendmesj}}%
    \@tempa % execute the constructed xdef
    \endgroup % restore normal catcodes
  }%
  \afterassignment\@tempa
  \toks\tw@=}
```

1.7 Reading functions

The \readline function gets one line of input from the user. Arguments are: #1 default to be used if the user response is empty (i.e., if the user just pressed the return/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 12.

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

Reset endline 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 unknowingly carries on 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 are both empty, we need something after #1 to keep \@car from running away, so we add an empty pair of braces.

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

\readChar reduces the user response to a single uppercase character. (This is useful to simplify testing the response later with \if.)

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

Reduce #2 to its first character, or the first character of #1, if #2 is empty. The extra braces {} are to prevent a runaway argument error from \@car if #2 and #1 are both empty.

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

The function \changecase uppercases or lowercases the replacement text of its second argument, which must be a macro. The first argument should be \uppercase or \lowercase.

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

We allocate a token register just for the use of \changecase because it might be used at a low level internally where we don't want to interfere with other uses of the scratch token registers 0–9.

```
\newtoks\@casetoks
```

A common task in reading user input is to verify, when an answer of a certain kind was requested, that the response has indeed the desired form — for example, if a nonnegative integer is required for subsequent processing, it behooves us to verify that we have a nonnegative integer in hand before doing anything that might lead to inconvenient error messages. However, it's not easy to decide how best to handle such verification. One possibility might be to have a function

```
\readnonnegativeinteger\foo
```

to do all the work of going out and fetching a number from the user and leaving it in the macro \foo. Another possibility would be to read the response using \readline and then apply a separate function that can be used in combination with \if, for example

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

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\tempcount#1%
  \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).

Argument #2 of \checkinteger must be a count register; #1 is expected to be 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 \scaninit 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 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 being tested. The test whether #1 is a decimal digit is similar in spirit to the test \if!#1! to see if an argument is empty (*The TEXbook*, Appendix D, p. 376).

```
\def\scanint#1{%
  \ifodd 0#11 %% is #1 a decimal digit?
%%   If so read all digits into \scanresult@
%%   with sign prefix
    \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
        \def\@tempa{%
          \scanresult@=-\maxdimen\endscan}%
  \fi\fi\fi
  \@tempa
}

\def\endscan#1\endscan{}

\newcount\dimenfirstpart

\newtoks\dimentoks
```

\scandimen is similar to \scanint but has to call some auxiliary functions to scan the various subcomponents of a dimension (leading digits, decimal point, fractional part, and units, in addition to the sign). The minimum requirements of TEX's syntax for dimensions are a digit or decimal point plus the units; all the other components are optional (*The TEXbook*, Exercise 10.3, p. 58).

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 decimal point.

```
\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 of the units.

```
    \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 evident here as we are able to apply it to only the test part of the conditional without running into brace-matching problems. (Compare the braces in this example to something like \message{\iffalse 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\endscan}%
      \else
        \def\@tempa{\scanunitsb xx\endscan}%
      \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. This extraordinary ploy should only be attempted by experienced TEX programmers possessed of the profoundest understanding of the language.

...Ha ha! Just kidding. Actually you simply have to realize that `\lowercase` and `\uppercase` are rather odd, then experiment to see what you can get away with.

```
\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`

This file requires `grabhedr.sty` and `dialog.sty`. If `grabhedr.sty` is not already loaded, load it now and call `\fileversiondate`, since it's too late to apply `\inputfwh` to *this* file. See the documentation of `\trap.input` in `grabhedr.doc`.

```
\csname trap.input\endcsname
\input grabhedr.sty \relax
\fileversiondate{menus.sty}{0.9q}{6-Jul-1994}
\inputfwh{dialog.sty}
```

2.1 Function descriptions

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

Defines `\foobar` as a function that puts the preliminary text, the menu lines (list of choices), and the after text on screen. Normal usage:

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

(See the description of `\readline` in `dialog.doc`.) In the various text parts all special characters have category 12 except for braces, as with `\mesj`. Note the recommended placement of the braces: no closing brace falls at the end of a line, except the very last one. Because of the special catcodes in effect when reading the final three arguments, a `^^M` or `%` between arguments would be read as an active character or category-12 character respectively,

instead of being ignored. But actually, after some rather difficult programming, I managed to make it possible to write just about anything (except brace characters) between the arguments and have it be ignored, so the recommended style is not mandatory. The first and last newline of each argument are stripped off anyway in order to produce consistent clean connections with `\menuprefix` etc.; see below.

Menu functions created by `\fmenu` are allowed optionally to have arguments, like functions created with `\fmesj` (from `dialog.sty`), so that pieces of text can be inserted at the time of use. This makes it possible for several similar menus to share the same menu function if there are only minor variations between them.

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

The text `\menuprefix` will be added at the beginning of each menu; `\menusuffix` will be added at the end. The text `\inmenuA` and `\inmenuB` will be added between the first and second, respectively second and third parts of the menu; their default values produce a blank line on screen. (But `\inmenuA` will be omitted if the 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 box-drawing characters in the standard PC DOS character set. A more detailed example in `menus.doc` is omitted here for the sake of brevity.

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

`\nmenu` and `\nxmenu` are like `\fmenu`, `\fxmenu` except that they automatically number each line of the middle part of the menu. (This allows menu choices to be added or deleted without tedious renumbering.) The first argument indicates the type of numbers to be used: `\alph`, `\Alph`, `\arabic`, `\roman`, `\Roman` (following LATEX). These are not yet implemented.

The function `\menunumber` (taking one argument) is applied to each automatically generated number. The default value is to add brackets and a space after:

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

but by redefining \menunumber you can add parentheses or extra spaces or what have you around each number. Internally a line of an autonumbered menu is stored as

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

\optionexec\answer

This is a companion function for \readChar and the menu functions: it checks to see if the answer is equal to any one of the characters ? Q X, and if so executes \moption? or \moptionQ or \moptionX respectively, otherwise executes

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

where C means the character that was read and \curmenu is a string identifying the current location in the menu system. (\optionexec pushes and pops \curmenu when going between menus, to keep it up to date.)

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.

\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 returns a boolean value; it is designed for use with \readline or \xreadline, to trap the special responses ? Q q X x before executing some conditional code. It returns true if and only if the replacement text of \answer is a single character matching one of those listed. This is used when you are prompting for a response that can be an arbitrary string of characters, but you want to allow the user still to get help or quit with the same one-character responses that are recognized in other situations.

2.2 Definitions

We start by using the \localcatcodes function from grabhedr.sty to save current catcodes and set new catcodes for certain significant characters, as explained (at more length) in dialog.sty.

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

\menuprefix is a string added at the beginning of each menu to pretty it up a little (or uglify it a little, depending on your taste). The length of the default string is 70 characters, not counting the two newline characters. By using \storexmesj we get embedded newlines corresponding to the ones seen here. [That is, except for the extra line break (where the newline character is commented out), needed to make this fit in *TUGboat*'s 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, #2 optional argument specifiers, #3 preliminary text, #4 list of menu choices, #5 following text. But at first we read only the first two because we want to change some catcodes before reading the others. The auxiliary function \fxmenub is shared with \fxmenu.

Because of the catcode changes done by \mesjsetup, newlines, spaces, or percent signs between the three final arguments will not be ignored. To get around this, we use the peculiar #{ feature of TEX, in intermediate scratch functions called \@tempa, to read and discard anything that may occur between one closing brace and the next opening brace. Token register assignments are used to read the arguments proper.

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

Before proceeding to define \fxmenub, we must deal with a subproblem. What we will have to work with is three pieces of text in the token registers \menufirstpart, \menuchoices, and \menulastpart, containing active ^^M characters to mark line breaks, including possibly but not necessarily ^^M at the beginning and at the end of each piece. What we would like to do, for each piece, is to remove the first ^^M, if there is one, and the last one, if there is one. The function \stripcontrolMs does this.

The technical details behind \stripcontrolMs found in menus.doc are skipped here for the sake of brevity, as they are unlikely to be interesting except to real TEX exegetes.

The argument of \stripcontrolMs is a token register. The text of the token register will be stripped of a leading and trailing ^^M if either or both are present, and the remainder text will be left in the token register.

```
\begingroup \lccode`\~=`\^^M
\lowercase{%
\gdef\stripcontrolMs#1{\expandafter\stripM
  \expandafter$\the#1$~$$\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\expandafter{\the#1#2##1#3}%
    \ifx\end##2\expandafter\@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.[4]

```
\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}%
```

[4] 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 ...

```
  \ifx\@tempa\@empty
    \let\nxa@\@gobble
  \else
    \addmenulines\menufirstpart
      \menutopline\endmenuline
    \let\nxa@\nx@
  \fi
```

If \menulastpart is empty, we don't add the separator material \inmenuB.

```
 \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 the new menu. #2 = begingroup or \xmesjsetup.

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

Temporarily \relaxify \menuline etc. in order to prevent their premature expansion if \xdef is applied.

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

Expanding analog of \fmenu.

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

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

2.3 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\expandafter{\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, as compared to \moptionQ, which 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 \fmesj here is to use % to comment out the initial newline, as the line break was needed only for convenient printing of this documentation within a narrow column width.

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

The function \optionexec takes one argument, which it uses together with \curmenu to determine the next action. The argument is expected to be a macro containing a single letter, the most recent menu choice received from the user.

Common options such as ?, Q, or X that may occur at any level of the menu system are handled specially, to cut down on the number of control sequence names needed for a csname implementation of the menus.

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

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

```
    \expandafter\let\expandafter\@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 back to %
the previous menu.
A response of X will get you entirely out of %
the menu system.
&\menusuffix%
Press the <Return> key ( Enter ) to continue:
}
```

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

\moptionhelp is the branch that will be taken if the user enters a question mark in response to a menu.

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

\expandafter\def\csname moption?\endcsname{%
  \moptionhelp}
```

The function \specialhelp can be used to provide a one-time alternate help message tailored to a specifc response given by the user. It defines the first argument (the macro containing the response) to contain ?, then redefines \menuhelpmesj to use the message text given in arg #2.

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

Init.

```
\def\specialhelpreply{}

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

Restore any catcodes changed locally, and depart.

```
\restorecatcodes
```

```
\endinput
```

Appendix
Miscellaneous support functions: grabhedr.sty

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{*filename*}

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

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

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.

```
\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`\^=7
\catcode`\_=8 \catcode`\^^M=5 \catcode`\"=12
```

Make @ a letter for use in 'private' control sequences.

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

A.1 Preliminaries

For \@empty, \@gobble, ... we use the LATEX names so that if grabhedr.sty is used with LATEX we won't waste hash table and string pool space.

Empty macro, for \ifx tests or initialization of variables.

```
\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 by grabhedr.sty, is needed by the principal customers of grabhedr.sty (e.g., dialog.sty).

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

To define \@@input as in LATEX we want to let it equal to the primitive \input. But if a LATEX format is being used we don't want to execute that assignment because by now \input has changed its meaning. And if some other format is being used it behooves us to check, before defining \@@input, whether \input still has its primitive meaning. Otherwise there's a good chance \inputfwh will fail to work properly.

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

This code shows a fairly easy way to check whether the meaning of a primitive control sequence is still the original meaning.

```
  \edef\0{\meaning\input}\edef\1{\string\input}%
  \ifx\0\1%
    \global\let\@@input\input
  \else
    \errhelp{%
Grabhedr.sty needs to know the name of the
\input primitive in order to define \inputfwh
properly. You might want to try to patch up the
problem by letting \input = \primitiveinput
before inputting grabhedr.sty.}
    \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.2 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.

```
\fileversiondate{grabhedr.sty}{0.9g}{6-Jul-1994}
```

Currently (July 1994) filehdr.el by default adds a string of equal signs (with an initial comment prefix) at the very top of a file header. This string must be scanned away first before we can start looking for the real information of the file header.

```
\xdef\@headerstart{%
  \@percentchar\@percentchar\@percentchar\space
  ==================================%
  ==================================}
```

The purpose of this function is just to scan up to the opening brace that marks the beginning of the file header body. Everything before that is ignored, not needed for our present purposes.

```
\gdef\@scanfileheader#1@#2#{\@xscanfileheader}
```

Throw in some dummy values of version and date at the end so that all we require from a file header is that the filename field must be present. The version and date fields can be present or absent, in any order, but the corresponding TEX variables \fileversion and \filedate will not get set properly unless the order is: filename, [...,] version, [...,] date.

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

This function assumes that filename, version, and date of a file are listed in that order (but not necessarily adjacent). It's possible for the version and date to be missing, or out of order, but in the latter case wrong values may be passed on to the \fileversiondate call. Trying to handle different orderings would be desirable but I haven't yet been struck by a suitable flash of insight on how to do it without grubby, time-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}{#4}{#6}%
}
```

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\@headerstart
    \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 first token will have its proper catcode because we did not change anything except the percent character. Otherwise, we still proceed with 'normal' input execution, but by making % active and defining it suitably, we can carry out further tests to see if the first file line has the expected form (three percent signs plus lots of equal signs).

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

A.3 Managing catcode changes

A survey of other methods for saving and restoring catcodes would be more work than I have time for at the moment. The method given here is the best one I know (other methods use up one extra control sequence name per file, or don't robustly handle multiple levels of file nesting).

The `\localcatcodes` function changes catcodes according to the character/catcode pairs given in its argument, saving the previous catcode values of those characters on a stack so that they can be retrieved later with `\restorecatcodes`. Example:

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

to change the catcode of `\@` to 11 (letter) and the catcode of " to 13 (active). In PLAINTEX you'd better be careful to use + instead of \+ in the argument of `\localcatcodes` because of the outerness of \+.

The way this function works is by using token registers 0 and 4 to accumulate catcode assignment statements: in `\toks0` we put the statements necessary to restore catcodes to their previous values, while in `\toks 4` we put the statements necessary to set catcodes to their new values.

```
\gdef\localcatcodes#1{%
  \begingroup \toks@{}\toks4 {}%
  \def\do##1##2{\ifx \end##1%
```

Finished processing the list; Take the accumulated contents of `\toks@` and add them as a new element at the top of the catcode stack. Adding the { } makes the new element easily poppable.

```
      \ifx\@empty\@catcodestack
        \gdef\@catcodestack{{}}%
      \fi
      \toks2 \xp@{\@catcodestack}%
      \xdef\@catcodestack{%
        {\the\toks@}\the\toks2 }%
      \xp@\endgroup\the\toks4 \relax
    \else
```

Add a catcode-restore statement at the beginning of `\toks@`.

```
      \edef\0{\catcode`\nx@##1=
        \the\catcode`##1\relax \the\toks@}%
      \toks@\xp@{\0}%
```

Add a catcode-setting statement at the end of `\toks4`.

```
      \toks4 \xp@{\the\toks4\relax
        \catcode`##1=##2\relax}%
      \afterfi\do
   \fi
  }%
  \do#1\end\relax
}
```

Initialize 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 this init is just for show since `\localcatcodes` is careful to add an empty element whenever 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.4 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, however: (1) the conditional remains open throughout the processing of everything in grabhedr.sty and the \fileversiondate statement, which makes any \else or \fi mismatch problems harder to debug; (2) if \undefined becomes accidentally defined the \ifx test will fail; (3) choosing the right control sequence to test against \undefined requires a little care.

In a situation where we know that the file to be input has had \fileversiondate applied to it, if it was already input, then we have a failsafe control sequence that we can test to find out whether the file has already been input — the name of the file. Assuming a standard form for the input statement (one that will work with either plain TEX or LATEX, and makes as few assumptions as possible), we can write a function that will trap input statements and execute them only if the given file has not yet been loaded:

```
\csname trap.input\endcsname
\input grabhedr.sty \relax
\fileversiondate{foo.bar}{1.2}{1993-Jun-07}
```

The function \trap.input scans for an input statement in canonical form and executes it if and only if the file has not yet been input (more precisely, if the control sequence consisting of the file name is undefined, which means that it has not had \fileversiondate applied to it). The canonical form that I consider to be the best is \input ⟨*full file name*⟩␣\relax. Having the \relax means that the input statement will not try to expand beyond the end of the line if \endlinechar is catcoded to 9 (ignore), as is done rather frequently now by progressive TEX programmers. The \relax would ordinarily render the space after the file name unnecessary, but I prefer leaving the space in to avoid interfering with redefinitions of \input to take a space-delimited argument that are occasionally done to achieve other special effects (see, for example, "Organizing a large collection of stylefiles", by Angelika Binding, *Cahiers GUTenberg*, numéro 10–11, septembre 1991, p. 175.) LATEX's argument form \input{...} cannot, unfortunately, be part of the canonical form if PLAINTEX compatibility is required.

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

End the group that encloses this entire file, and then call \endinput.

```
\endgroup
\endinput
```

◇ Michael Downes
49 Weeks Street
North Smithfield, RI 02895
U.S.A
mjd@math.ams.org