Carrying on Dialog with TEX

Michael J. Downes

November 1994; reprocessed January 2013

Contents

1 Introduction

A common task in any programming language is to send a question to the user, and to read (and act on) the user's response. In TEX, this usually involves the \read, \message, and \write commands. The use of these commands, however, is beset at every turn by odd hindrances and technical stumbling blocks, so that even experienced macro writers, faced with an application that requires a bit of dialog, usually find it troublesome to make that bit of dialog good-looking, reliable, convenient for the user, and tolerant of typical human mistakes such as minor mistyping in a response. The purpose of this article is to analyze the capabilities that TEX has for dialog and survey all the best relevant macro-writing techniques that fall within the scope of my experience and research.[1]

In *The TEXbook*, near the end of Chapter 20, Knuth writes

> It's easy to have dialogs with the user, by using \read together with the \message command

and there follows a brief example involving reading the user's name into a macro \myname. It's clear from this passage that Knuth means, by *dialog*, ordinary communication between computer programs and users at the early-1980s level of technology: printing character-based information on a video screen, displaying menus, asking questions, and handling user responses. (Speech recognition and voice synthesis are not part of the picture for most TEX users, yet.) For the purposes of this article, I define *dialog* as any communication between TEX and the user that takes place while TEX is running. Forms of communication that do not take place while TEX is running are excluded: for example, the black box that TEX prints to indicate an overfull line in a paragraph is useful information, but not dialog because the communication occurs after TEX has stopped running.[2] On the other hand, the Overfull \hbox message printed on screen whenever TEX adds a black box to the current paragraph *is* dialog, because it does occur while TEX is running.

Since the entire section of *The TEXbook* where Knuth's dialog example appears is marked off with double dangerous bend signs, it seems that Knuth didn't intend his words "It's

[1] This is an overhauled and amplified version of my paper "Dialog with TEX" in *TUGboat*, vol 12 no 3, Part 2, December 1991 (proceedings of the 1991 TUG conference in Dedham, Massachusetts).

[2] This distinction is blurring, however, with the advent of software like Blue Sky Research's Lightning Textures.

easy" to be taken completely literally—particularly when we look at the next thing in that section, Exercise 20.18, which reads,

> The `\myname` example just given doesn't work quite right, because the ⟨return⟩ at the end of the line gets translated into a space. Figure out how to fix that glitch.

That line-ending space is only one of a number of complications that can hamper the efforts of macro writers to write dialog for practical applications. In fact, the word *easy* is far from the first adjective that comes to mind when I remember my own early attempts at writing dialogical macros. Though my thrashing and floundering for the most part took place behind the scenes, invisible to others, it eventually reached the point of threatening my secret belief that I was a hot-shot macro writer. That spurred me to start paying special attention to anything related to the idea of dialog in TEX, and accumulating scraps and pieces of assorted useful techniques. This article is more or less a survey of what I've learned so far. Sections 2 and 3 review the functions TEX provides to support dialog; sections 4 and 5 discuss common difficulties and how to handle them.

1.1 Terminology

Rather than assume all readers know well enough the meaning of terms like *primitive*, *token*, or *control word* that will be bandied about hereinafter, I offer a quick review of some standard TEX terminology, to aid those who want it, and to be skipped by the rest of you.

A TEX command is either a *control sequence*—a string of characters starting with an *escape character*—or a single *active character*, such as `~`. The usual escape character is the *backslash*, `\`. A control sequence that consists of a backslash plus one nonletter character is called a *control symbol*; a control sequence that consists of a backslash plus one or more letters is called a *control word*. Spaces are ignored after a control word, but not after a control symbol. A control sequence is either a *TEX primitive*—a command built into the TEX program—or a macro: a composition of primitives or other macros, defined by the user or by a *macro package* such as LATEX or PLAIN TEX. A control sequence may require one or more following *arguments*; an argument is a piece of text that is grabbed up by the control sequence in order to do something with it. Arguments typically are enclosed in *curly braces* `{` and `}`. For example, the command `\sqrt` is a macro with one argument, and is used thus: `\sqrt{x^2+y^2}`, producing the printed output $\sqrt{x^2+y^2}$.

Macros are *expandable*; some primitives are expandable, more are not. An expandable control sequence will be replaced by its expansion if it is used inside the argument of a `\message` command, or anywhere else where TEX is in an expansive mood (`\write`, `\errmessage`, `\edef`, `\xdef`, `\mark`, `\special` [*The TEXbook*, p216]).

The term *parameter* is used to mean a numeric or dimensional variable such as `\hyphenpenalty`, `\hsize`, or `\baselineskip`. A *token* is eithera character (with associated catcode) or a control sequence, *after* it has been read by TEX from some file and entered into TEX's active memory. Character tokens can only have category codes 1–4, 6–8, or 10–13; there's no such thing as a 'character token' with category code 0, 5, 9, 14, or 15: those catcodes only control the process of creating tokens, they aren't designed for permanent association to a token.

Under normal circumstances, each line in a file is understood by TEX to have a `^^M` character, ASCII 13, at the end of it, even if your text editor actually puts some other character, or no character, at the end of a line when you press the RETURN or ENTER key.

1.2 Basic dialog principles

It's not hard to identify a number of principles that make for good dialog:

1. When asked a yes/no question, users should be able to enter `y`, `yes`, or even `ye`, in lowercase, uppercase, or even mixed case, and have the answer understood to be "yes".

2. For any menu or question, a default answer should be provided (when this makes sense), and the default answer should be made as easy as possible to select.

3. Users' answers should be repeated back to them, to allow them to verify that the program's impression of the answer entered by the user is indeed correct.

4. Users should be given a chance to undo mistakes, e.g., by going back to a specified point earlier in the dialog and starting over from there. For example, it shouldn't be necessary to stop TEX and restart just to fix a typing error.

5. When practical, users' answers should be checked to make sure they're not nonsense; for example, if a program requests an integer, it should check the response to make sure the user didn't enter something else entirely, rather than assume an integer was entered and start to perform operations on it. In TEX this would create a risk of losing control to low-level errors such as `Missing number, treated as 0` or `Arithmetic overflow`.

6. Information given to users should be provided in the "best possible form", where the meaning of "best possible" must be determined by common sense from the circumstances of a particular application and the targeted user group. For example, a straightforward use of the `\the` command to report the value of a TEX dimension parameter such as `\vsize` to the user will produce the value in points, down to five or six decimal places. It will normally be more useful to report the value rounded to the nearest whole point, or to report it in picas, inches, or centimeters—whatever is most convenient for the user. A typographical designer or compositor would probably prefer picas, while someone with little knowledge of typography would probably prefer inches or centimeters.

Table A: Commands that can be used for sending messages

Command	Example	Result
\message	\message{Hey you}	... Hey you ...
\write	\immediate\write{Hey you}	... Hey you ...
\errmessage	\errmessage{Hey you}	! Hey you. l.217 \errmessage{Hey you}
\show	\show\footnote	> \footnote=macro: ->\@ifnextchar [{\@xfootnote } ... l.218 \show\footnote
\showthe	\showthe\textwidth	> 570.93257pt. l.219 \showthe\textwidth
\showbox	\showbox 0	> \box0= \hbox(0.0+0.0)x15.0 ! OK. l.220 \showbox0
\showlists	\showlists	### vertical mode entered at line 0 ### current page: \glue(\topskip) 3.75 ... total height 403.47491 plus 14.64996 minus 8.77498 goal height 751.60756 prevdepth 0.0, prevgraf 2 lines ! OK. l.221 \showlists

2 TEX's Message-Sending Capabilities

Table A lists the various means in TEX for sending messages to the user.

Although it could be argued that the token register \errhelp is another way of sending a message, it is excluded from Table A on the grounds that it is passive rather than active, unlike the other commands listed. To put it another way, \errhelp is merely a storage area associated with \errmessage, where auxiliary text can be placed; the user won't ever see \errhelp except by way of \errmessage.

2.1 The \message primitive

The \message command is a TEX primitive that prints its argument on screen. If the current screen position is not at the beginning of a line, TEX will add a blank space at the beginning of the message text to separate it from the preceding material—except that if there isn't enough room on the current line to fit the entire message text, then TEX will go to the next line before starting to print the message, and not add an extra blank at the beginning. The maximum length of message lines is controlled by the constant *max_print_line*, which is compiled into TEX; the normal value is 79. (In a windowing environment the width of the current window may also affect the maximum length of message lines.)

Thus one way to force a message to start on a new line is to add lots of \space's at the end. But a better way to start a message on a new line, or break up a long message into lines, is to indicate line breaks with the current \newlinechar character. For example, we can set the newline character to be + and use it in a message as follows:

```
\begingroup \newlinechar=`+[3]
\message{+This is a+three-line+message ...}
\endgroup
```

which produces on screen (regardless of the length of any immediately preceding message)

```
This is a
three-line
message ...
```

In any one message, a given character can either produce newlines, or represent itself, but not both. As a consequence, if we wanted a plus character in a message to actually print on screen instead of causing a line break, we would have to set \newlinechar to some other value before sending the message. From this knowledge it's a short step to the insight that for general message-sending purposes it would be convenient to set \newlinechar to the character that is least likely to be needed in a message text. The nonprinting ASCII characters in the range 0–31 are obvious candidates.

[3] Use of \+ instead of + here would normally be recommended but the outerness of \+ in PLAIN TEX makes this an exception.

But here we encounter an inconvenient idiosyncrasy of TEX: A control character—such as control-J, or `^^J` (using TEX's double caret notation), which is the default value for `\newlinechar` in $\mathcal{AMS}$-TEX and LATEX—doesn't ever produce line breaks in a `\message`, even if it is currently selected as `\newlinechar`.[4] Instead, it will always be printed as three characters using the double caret convention.[5] Therefore, if you want to use `\message` as your normal message-sending function, you should choose one of the seldom-used printable characters as your default `\newlinechar`. One possibility would be to use the double quote character for this purpose, since single quote characters can normally be substituted for double quote characters in message texts. `Testfont.tex` [Knuth, 1986c] uses the `@` character.

Unlike `\message`, however, the `\write` command is capable of using a control character as a newline character (see §2.2). By using `\write` for multiline messages, and making `^^J` the default newline character, $\mathcal{AMS}$-TEX and LATEX avoid taking any of the printable characters out of circulation for dialog purposes.

If no line breaks are indicated in a `\message` that is longer than *max_print_line*, TEX will introduce arbitrary breaks at the screen column equal to *max_print_line*, which usually means

```
random brea
ks in the m
iddle of wo
rds.
```

Those of you who use `\tracingmacros` will have noticed that its output also has line breaks like this. Which can make it rather difficult to search for instances of a given string in the trace log; to find all instances of `xyz` you need a regular expression something like

```
x\n?y\n?z
```

(`\n` meaning 'newline', `?` meaning 0 or 1 occurrences of the preceding subpattern).

2.2 The `\write` primitive

The `\write` command, like `\message`, just prints a message. But the message doesn't necessarily appear on screen, because communication with the user is not the purpose for which `\write` was originally designed: Its initial purpose was to send index or table of contents information, including the associated page number, to an auxiliary file for later processing. Because this kind of use is closely linked to page numbering, `\write` commands on the current page are normally saved up to be executed when the page is actually shipped out, after the page break has been determined. If such postponement is not wanted, `\write` must be used with the `\immediate` prefix.

In order to allow intersequential writing to different output files, the `\write` command takes an extra first argument, a number between −1 and 16 inclusive, to indicate the output file to which the text should be sent. Output files 0–15 can be associated with a particular file on your system by the `\openout` command; output file −1 is the TEX log file, and output file 16 is the user's terminal screen (echoed in the log file as well).

Line breaks in the argument of a `\write` command can be obtained by inserting `\newlinechar` characters; unlike `\message` (§2.1) and `\errmessage` (§2.3), `\write` will always start a new line for each newline character, even when it is a control character such as `^^J`. Also, the text of a `\write` command always starts on a new line and finishes on a new line. The existence of the final newline may be observed in the on-screen result of a `\message` following a `\write`: the message text will always start on the next line regardless of the total length of the `\message` and `\write` texts, whereas a `\message` following another `\message` or one of TEX's internally generated messages (such as input file names) will not start on a new line unless there isn't enough room remaining on the current line.

Corollary: If you prompt the user for some input and you want the user's input to appear on the same line as the prompt text, use `\message` instead of `\write` to send the prompt text—or at least the last line of the prompt text.

Nonimmediate `\write` messages

Sometimes it's useful to leave off the `\immediate` prefix of a `\write` command even when not writing information to an index file or table of contents file: For instance, if you are working on page breaks in a long document and want to find out, without previewing or printing, if a nonforcing pagebreak command had the effect that you wanted, you could insert a nonimmediate `\write16` just before and just after the intended page break:

```
\write16{Before the attempted pagebreak.}
\penalty-9999
\write16{After the attempted pagebreak.}
```

The message from a nonimmediate `\write16` will appear before the closing `]` of the `[]` pair that enclose the relevant page number. So if all went well, one of the above messages will appear with one page number and the next message with the next page number, like this:

```
[4] [5
Before the attempted pagebreak.
] [6
After the attempted pagebreak.
] [7] [8] [9] ...
```

In producing this article (using LATEX) I had some trouble getting good placement for the floating tables and examples; to help me experiment, I added some code that would print on screen the page numbers. At the beginning of Example 2 (for example) there is a line that says

```
\write16{Example 2: Page \thepage}%
```

The `\immediate` prefix must be omitted in order to get the page number correct.

[4] Or rather, it didn't up until version 3.141 or so of TEX. See also the mention of emTEX's `/r` option (which allows you to use control characters for output purposes) in §5.11.

[5] This suggests the following experiment: set `\newlinechar='\^` and send a `\message` containing a `^^J` character.

2.3 The \errmessage primitive

The \errmessage command prints its argument on screen, starting on a new line, with an exclamation point and a space added at the beginning, and a period added at the end. For example, \errmessage{Surprise} produces

```
! Surprise.
```

on screen. \errmessage also shows the *current context*, which means the current line from the current input file, along with the line number, and additional information if there is any (such as the surrounding parts of current macro expansions).[6] So the Surprise error message would show additional information on screen. Suppose we define

```
\def\test{\errmessage{Surprise}\relax}
```

Then the additional information will look something like:

```
! Surprise.
\test ->\errmessage {Surprise}
                              \relax
l.454 \test
           and some more text.
?
```

This example may be interpreted as follows: Line 454 of the current file consists of

```
\test and some more text.
```

The line break in the context listing means that TEX is processing \test and has not yet started to typeset the word 'and'. Above the l.454, the expansion of \test is shown. The line break after {Surprise} indicates that TEX has not yet executed the \relax command.

The behavior of \errmessage with respect to newline characters and control characters is the same as for \message—i.e., \errmessage will start a new line for each \newlinechar in its argument [unless the current value of \newlinechar is outside the visible ASCII range 32–126 and the version of TEX is less than 3.141].

At the end of an error message, the user is presented with a question-mark prompt, and a choice of several possible responses. These will be discussed later in the section on TEX's capabilities for receiving user input (§3.2).

2.4 The \show and \showthe primitives

The \show command, used for showing the current meaning of a control sequence (or indeed of any token), is rather similar to the \errmessage command in what it produces on screen. The prefix is a greater-than character instead of an exclamation point. Here's the result of \newcount\C \show\C:

```
> \C=\count78.
l.1 \newcount\C \show\C

?
```

[6] If the parameter \errorcontextlines is set high enough.

As with \errmessage, TEX displays the surrounding context of a \show command; it also offers the same question-mark prompt with the same range of possible responses (well, almost—the H option only gives a generic help message about \show, not specific help about the item being shown.).

The \showthe command is like \show, but is applied to certain kinds of things such as count registers and token registers, that have not only a meaning but also a current value. For instance, here's the result of \C=5 \showthe\C, using the counter defined above:

```
> 5.
l.3 \C=5 \showthe\C

?
```

2.5 The \showbox and \showlists primitives

The commands \showbox and \showlists are similar to \show in what they produce on screen (see Table A). Because of their specialized nature they don't ordinarily have much application in dialog between TEX and the user.

2.6 Piggybacking

Many messages printed on screen by TEX do not involve any of the commands listed in Table A. These other messages are emitted directly by TEX, outside the control of the macro writer. However, with a little imagination, you can often find ways to attach useful information to those "inaccessible" messages. This is what I mean by *piggybacking*.

For example, whenever TEX inputs a file, a message is printed on screen containing the name of the file, enclosed in parentheses. So one way to send a short message would be to create an empty file whose name was equal to the desired message, and then input the file.

File name messages, apart from the parentheses, behave the same as messages produced by the \message command: in particular, a file name message will be appended to the current line, with a preceding space, unless the length of the message (including the two parentheses) will cause it to cross the *max_print_line* boundary.

If your computer system allows longer file names you could actually get pretty fancy with a filename message. For example, on a Unix system suppose you have a file named test.tex whose contents are

```
\newwrite\msgfile
\immediate\openout\msgfile=Fred.your.fly.is.open
\immediate\write\msgfile{\relax}
\immediate\closeout\msgfile
\input Fred.your.fly.is.open
\end
```

When test.tex is processed by TEX the screen output will be something like:

```
This is TeX, C Version 3.1 (format=plain 91.1.4) ...
(test.tex (Fred.your.fly.is.open) )
No pages of output.
```

However, getting spaces in the message would be problematic since TEX treats a space as a file name terminator. And

if your message doesn't include a period, TEX is likely to add `.tex` at the end of the file name when `\openout` is invoked. Not to mention that this method would quickly lead to inconvenient file clutter since TEX can't delete files, only create new files or change the contents of pre-existing ones.

Or consider the count registers 1–9; if any of these is nonzero, its value is reported on screen whenever TEX ships out a page. One way of using this feature might be to report the accumulation of index terms for a document, by having each index command increment count register number 1:

```
\countdef\indexcount=1
\def\index{...
  \global\advance\indexcount 1 ...}
```

Incrementing count 1 like that would result in TEX displaying on screen something like

```
[1.2] [2.7] [3.14] [4.15] [5.27] [6.38] ...
```

instead of the more usual

```
[1] [2] [3] [4] [5] [6] ...
```

as each page is shipped out.

These two examples don't seem extremely practical—the total number of index terms reported in the latter example won't necessarily be correct page for page, until the end of the document, because ordinary means for incrementing the counter are immediate in their effect rather than synchronized with the `\write` commands used for creating the index file—but the point is to realize that the messages coming out of TEX's innards aren't totally beyond reach, and they can sometimes provide a better way of achieving a given result than ordinary methods. A couple of better-known examples, from the hands of Donald E. Knuth, can be found in the `\showhyphens` command and in `hyphen.tex` at the point where `\patterns` is called.

The `\showhyphens` command

The `\showhyphens` command (defined in `plain.tex`) works by exploiting TEX's messages about underfull boxes. When an underfull line of a paragraph is reported, TEX prints on screen the elements of that line, including any discretionary hyphens inserted by TEX while attempting to find good line breaks. The key insight for thinking up the `\showhyphens` command is to realize that if you can typeset a one-line paragraph, and make sure that the line is underfull, then any word in that line will have its hyphenation points displayed on screen. That's exactly what Knuth defined `\showhyphens` to do:

```
\def\showhyphens#1{\setbox0\vbox{\parfillskip\z@skip
  \hsize\maxdimen \tenrm \pretolerance\m@ne \tolerance\m@ne
  \hbadness0\showboxdepth0\ #1}}
```

The settings of `\parfillskip` and `\hsize` ensure that the paragraph will be all on one line, and underfull. (For extra bullet-proofing,

```
  \leftskip\z@skip \rightskip\z@skip
```

should probably be included too.)

The switch to font `\tenrm` makes sure—or at least reasonably sure—that the current font is not one for which hyphenation has been inhibited by setting `\hyphenchar` to an out-of-range value. The settings of `\pretolerance` and `\tolerance` ensure that hyphenation points will be added (in making up a paragraph, TEX normally tries first to get by without adding hyphenation points, if it can find good line breaks using only the stretchability and shrinkability of interword glue). The setting of `\hbadness` ensures that an `Underfull \hbox` message will be sent (otherwise, if the surrounding environment had `\hbadness` = 10000 when `\showhyphens` was called, the message would be suppressed). The setting of `\showboxdepth` limits the information in the message to top level; otherwise compound structures like accented letters or special composites (e.g. `\AA`) would be shown in full detail—more detail than the user normally wants to see. And finally the `\␣` command forces entry into horizontal mode and, more importantly, provides a glue item before the first word, without which it would not be hyphenated (see the rules by which TEX looks for hyphenatable words, *The TEXbook*, Appendix H).

Note that the values of `\language`, `\lefthyphenmin`, and `\righthyphenmin` are inherited from context; and this is probably what you want for the `\showhyphens` command (perhaps indeed the current font should also be inherited from context instead of being set always to `\tenrm`).

Using error context to send messages

The standard `hyphen.tex` containing U.S. English hyphenation patterns has a comment after the `\patterns` command:

```
\patterns{ % just type <return> if you're not using INITEX
```

Ordinarily the macro writer can't use comments to communicate with the user, because comments within the text of a macro are discarded by TEX as the macro is defined. The beauty of the comment in `hyphen.tex` is that it appears precisely when needed, because of the way TEX displays context with error messages: if you `\input hyphen.tex` when not using INITEX, TEX will give an error message when it encounters the `\patterns` command, and as usual, will show the context around the point of the error, like this:

```
! Patterns can be loaded only by INITEX.
l.2 \patterns
              { % just type <return> if you're not using INITEX
?
```

Since learning this technique, I've had occasion more than once to apply it in similar situations. One such application had to do with the LATEX circle fonts. I had an assignment to create a LATEX documentstyle whose installation procedures involved rebuilding the LATEX format file, which meant calling for the `.tfm` files of the circle fonts. However, there was at that time (1990–1991) a bit of confusion surrounding the names; the trend among distributors of TEX appears to be away from the original names `circle10` and `circlew10` and toward the names `lcircle10` and `lcirclew10`. The `.tfm` files are the same under either names, but I had to deal with

Table B: Receiving

Method	Prompt displayed by TEX
`\read`	`\controlseq=`
error message interaction	`?`
'show' message interaction	`?`
input file not found	`Please type another input file name:`
output file not writable	`Please type another output file name:`
interrupt key	none

the possibility that some users of the documentstyle I was working on would have the fonts under the older names, while others would have them under the new names.

My solution was to use the newer names `lcircle10` and `lcirclew10` and put comments on the same lines as the font assignments, so that the comments would appear to the user if TEX were unable to find the `.tfm` files and emitted an error message.

```
\font\tencirc=
lcircle10\relax% Type x to exit; see sei.prl for further info.
```

Additional comments in the file `sei.prl` explained how to change the font names to their older variants. The `\relax` is necessary so that TEX won't proceed to the next line, bypassing the comment, in the process of looking for a modifier such as `scaled` or `at`. And the reason for the line break after the equals sign is to fit a few more characters in the comment, which would be elided by TEX if too long.

3 Ways for TEX to receive messages from the user

Table B lists the various means in TEX for reading input from the user. The primary input facility is the `\read` command; the others are special cases applicable only under limited circumstances.

The purpose of getting input from the user is essentially always the same: to give the user an opportunity to change the outcome of the TEX run, which would otherwise be totally predetermined by the contents of the files read by TEX. (Well, and perhaps by a few system variables such as `\time`, `\day`, etc.)

3.1 The `\read` primitive

The form of the `\read` command is

```
\read 16 to \controlseq
```

where the number 16 is the input stream number, which might be any value from −1 to 16; a number between 0 and 15 would indicate reading from a file stored on disk, while 16 and −1 indicate reading from the user's keyboard. `\controlseq` can be any control sequence chosen by the macro writer. If the input stream number is 16, TEX will display a prompt of

```
\controlseq=
```

If the input stream is −1, this prompt will be omitted. In either case, a `\read 16` or `\read -1` command should normally be preceded by a message that lets the user know what kind of input to provide.

The action of the `\read` command is similar to that of the `\def` primitive. Both of them create a new macro containing an unexpanded token list, which must contain balanced braces. The balancing required in the response to a `\read` command, however, is slightly different than for a `\def`; as long as there an equal number of opening and closing braces, it doesn't matter how they're distributed—TEX will be perfectly happy with the response

```
a}}}b{{c}{{
```

as can be verified using `\show\answer` after reading the above string into `\answer`.[7] TEX will read more than one line, if necessary, if the first line contains an unmatched brace. This is a useful property if you *want* to read more than one line at a time, as is sometimes the case: write the information in the form `{ ... }`, and you can have as many lines between the curly braces as you want.

TEX always reads line by line, rather than character by character; unlike some other programming languages, TEX provides no way to read a single character and act on it immediately; the user must always press the RETURN key before anything will happen.

3.2 Error recovery

As mentioned earlier, after an error message TEX presents the user with a question-mark prompt. Typing a second question mark in reply to the prompt will cause TEX to list the options that are available:

```
! Error message.
   ...
? ?
Type <return> to proceed, S to scroll future error messages,
R to run without stopping, Q to run quietly,
I to insert something, E to edit your file,
1 or ... or 9 to ignore the next 1 to 9 tokens of input,
H for help, X to quit.
?
```

Choosing the `H` option causes TEX to print a help message containing additional information related to the error message. If the error message was generated using `\errmessage`, then the help message will consist of the current contents of the token register `\errhelp` (which should be filled with something useful by the macro writer, immediately before the call to `\errmessage`). Otherwise,

[7] See also §5.7. I questioned the behavior of `\read` with respect to extra closing braces and outer control sequences in a letter to Donald Knuth (December 1991); his response was to make a change in TEX (version 3.141?) that causes a `\read` operation to terminate decisively if an extra `}` or outer control sequence is encountered. So backwards balancing of braces will no longer be permitted.

Example 1: Using error interaction possibilities to get past a potentially bad error: a missing \\ before an \hline in a LaTeX tabular environment.

```
! Misplaced \noalign.
\hline ->\noalign
                  {\ifnum 0=‘}\fi \hrule...
l.120 \hline

?
```

Let's see what the help information is:

```
? h
I expect to see \noalign only after the \cr of an alignment.
Proceed, and I'll ignore this case.

?
```

Let's try skipping one token to verify what LaTeX is going to process next:

```
? 1

\hline ->\noalign {
                   \ifnum 0=‘}\fi \hrule...
l.120 \hline

?
```

All right, the opening curly brace has just gone by. We need to insert the \\ that was forgotten, and also replace the two tokens \noalign and { that have slipped by.

```
? i\\ \noalign{
```

And now LaTeX will be able to continue with the rest of the table.

for a built-in error message, the corresponding built-in help message (from tex.pool) is displayed.

The intended use of the insertion (I) and token deletion (1...9) options is for error recovery; after you look at the context of an error, you may be able to temporarily repair the damage and continue processing the remainder of the document, by removing some tokens and/or inserting others. Then if any other errors are uncovered later, they can be fixed at the same time as the first error, instead of requiring a second TeX run to find them. Example 1 illustrates this. You can actually delete up to 99 tokens at a time in all implementations of TeX that I know of, even though the help message suggests that 9 is the maximum.

The token deletion option can also serve as the basis of a crude menu facility. The idea is to use \errmessage to present a menu with choices labeled by numbers. If the user responds by entering, say, 3 to choose item 3, then we must arrange things so that after \errmessage does its normal thing of deleting three tokens, the following token that was not deleted should do something to ensure that item 3 will be selected. This isn't too hard if we use something like the following sequence:

```
\errmessage{...}\0\1\2\3\4\5\6\7 ...\stop
```

Clearly the deletion of three tokens after the error message will leave \3 as the next command to be executed, and so if we define \3 to do the right thing and then skip over the following \4\5... corresponding to unselected menu choices, we get what we want.

If the user just presses RETURN without entering a number, it will be \0 that is executed—therefore \0 should be defined to produce the default selection.

One thing that makes me call this crude is the fact that TeX pauses after any token deletion operation instead of barging ahead. This means that if the user chooses anything other than the default selection, they will have to press RETURN twice after typing the number, instead of just once. Some other crudities are introduced by the bits and pieces of an \errmessage that cannot be suppressed. These include:

- Exclamation point and space at the beginning of the error message.
- Period at the end of the error message.
- The expansion of the current macro (if the error message is contained in a macro), on two lines with the line break immediately after the token that was last processed by TeX. This includes, at the beginning, the macro name followed either by ->, if the first part of the expansion text is relatively short, or by ellipsis dots ... plus the tail end of the first part of the expansion text.
- The current line of the current file, on two lines with the break immediately after the token that was last processed by TeX.
- The question mark and space prompting the user for a response.

I would have hoped that setting \errorcontextlines to 0 or −1 would cause the 'innermost' two lines of the error context to be suppressed but apparently there is no value for \errorcontextlines that will suppress them. The following example illustrates all of the nonsuppressable parts.

```
! Error.
\CM ->\errmessage {Error}
                          \CM
l.38 \CM
       % A comment in the file, line 38
? x
```

Except for the ellipsis dots alternative, that is. Here's how that looks:

```
! Longer message text, forcing elision.
\CM ...sage {Longer message text, forcing elision}
                                                  \CM
l.38 \CM
       % A comment in the file, line 38
?
```

Although the exact number can vary (depending on how your particular version of TeX is configured at compile time), the maximum length of the first line of context is normally between 40 and 50 characters, and if the expansion text would

make the line longer than this, it is truncated at the beginning and the ellipsis dots are inserted.

Very well then. Since we have `!`, `.`, and `->` or `...` in the first two lines (the lines that will be nearest our menu text), and we cannot get rid of them, the next best thing is to camouflage them. One possibility is setting `\newlinechar=‘\!` just before sending the `\errmessage`, so that the `!` character will cause a blank line rather than printing on screen, and then putting a bunch of periods in the menu text to camouflage the other periods.

Appendix C exhibits `fontmenu.tex`, a more extensive working-out of this idea in which I tried to pound the recalcitrant `\errmessage` into the most presentable shape possible, using every macro hack I could think of.

3.3 Show message ‘recovery’

After a `\show`, `\showthe`, `\showbox`, or `\showlists` command, TEX offers a question-mark prompt, and the same menu of options as after an error message. There is only one slight difference: The `H` option provides no access to the `\errhelp` token register; only a generic help message about the `\show...` commands is available.

3.4 “Please type another input file name:”

When you see this prompt, displayed by TEX when it’s unable to find an input file, you have strayed into one of the less friendly byways of TEX. If you can’t think up a good file name to give as an answer, you could get stuck in an endless loop. Even simply pressing the RETURN key causes TEX, on most computer systems, to look for a file called ‘`.tex`’ which will most likely be nonexistent. Power users know that on many systems you can enter a file name of ‘`nul`’ to cause TEX to read in an empty file named `nul.tex`. But it is precisely power users who are likely to know other ways of getting past this prompt (for example, on some systems typing a `^^Z` or `^^D` character also does something useful), and it is precisely the users with no other clue what to try next who won’t know about `nul.tex`.

It seems that it would be useful for all standard distributions of TEX to provide files named `.tex`, `h.tex`, `help.tex`, and `?.tex` in the standard TEX inputs path, so that when users type `h` or `help` or `?` or just press RETURN, they will get the corresponding file. (Unfortunately, most operating systems don’t permit the question mark in file names, which means that only the other three files will normally be viable.) `Help.tex` and its clones could contain something as simple as:

```
\errmessage{Type ? to see your options; X to exit}
```

which would give the user access to the full menu of normal error recovery options.[8]

A menu trick

The `Please type another input file name` prompt is used to implement a sort of menu in the file `lfonts.new` of the Mittelbach/Schöpf font selection scheme (LATEX version), which has a statement `\input fontdef.tex`, where the file `fontdef.tex` is normally missing, intentionally, and the user is supposed to substitute another file name such as `fontdef.ori` or `fontdef.max`. The idea of comments designed to appear through TEX’s display of error context (§2.6) could be used to good advantage here, to tell the user what other file names are likely candidates:

```
\input fontdef.tex % Try fontdef.ori or fontdef.max
```

In this kind of application, additional help information in a message preceding the `\input` statement could also be useful.

3.5 Interrupt key

The interrupt key is a key (system-dependent, but `^^C` on many systems) that allows you to interrupt TEX when it is in the middle of doing something else. The normal reaction of TEX when the interrupt key is pressed is to print a message

```
! Interruption.
... % current context
?
```

in the same form as an error message, complete with a question-mark prompt, with the usual options available to the user.

It’s stretching the concept a bit to claim that the interrupt key is a way for users to send information to TEX; it has the flavor of the story about the farmer who had to whack his mule over the head with an axe handle “just to get his attention”. When you interrupt TEX you could easily find yourself in the middle of some complex macro where it would be inadvisable to do anything except use the `X` option to exit. However, this in itself is frequently useful.

4 Stumbling blocks in the use of `\write` and `\message`

4.1 Line breaking

As explained elsewhere (§2.1) it is impossible to use a control character as a newline character in the argument of a `\message` or `\errmessage` command. *[Note (30-Oct-1993): that’s no longer true, as of TEX version 3.141.]* LATEX and AMS-TEX use `\immediate\write` instead of `\message` in their all-purpose message macros (`\typeout` and `\W@`), which allows them to have `^^J` as the default newline character, thus leaving all of the printable characters usable in message texts. (If a given character is the current `\newlinechar`, there really is no way for TEX to print it on screen. Try setting `\newlinechar=‘\(` and see what happens to the file name messages for input files.)

There is a minor inconvenience with the use of `^^J` as the newline character. Under current conditions (as of 1993), it is usually wise to limit the length of all lines in a macro file to 72 characters, in order to avoid truncation problems that occasionally occur in, e.g., electronic mail transmission. When constructing a long message, if you leave `\endlinechar` at its normal value of `^^M` and put `^^J`’s at

[8] This idea is discussed at greater length in a recent *TUGboat* article of mine (to appear, late 1994).

the end of each message line, you get four extra characters at the end of each line, three for the `^^J`[9] and one for a percent sign to eliminate the space that would otherwise be produced on screen at the beginning of the next message line, by the `^^M`. This means that the effective limit on the length of each message line is 68 characters rather than 72.

But the clutter of four extra characters at the end of a line can be avoided by temporarily assigning `\endlinechar = \newlinechar` while a message is being constructed. This is assuming, however, that a useful value of `\newlinechar` has been established and that the same value will be in effect when the message is sent. If the construction and sending are simultaneous, the code can be as simple as this:

```
\begingroup \endlinechar=\newlinechar
\immediate\write16{Line 1
Line 2
Line 3
}%
\endgroup% this percent sign is necessary
```

And then the actual message text in each line can run to the full 72-character length if necessary. If a useful value of `\newlinechar` has not been established (e.g., when using PLAIN TEX without modifications), then rewrite the first line above as:

```
\begingroup \newlinechar=\endlinechar
\catcode\endlinechar=12 % Make it 'other'
```

Here we make the reasonable assumption that `\endlinechar` has its normal value of 13 (`^^M`); even this assumption could be done without by adding the statement `\endlinechar = 13` before the other assignments.

The percent sign is necessary after the `\endgroup` in the first variant, because the endline character for a given line is added when the line is initially read, before TEX begins to execute the line. Therefore by the time TEX processes the `\endgroup` and reverts to the previous value of `\endlinechar`, it is too late to prevent getting a newline character at the end of the current line.

For the same reason, a percent sign is not needed after the line that contains

```
\endlinechar=\newlinechar
```

The endline character for that line has already been tacked on by TEX and cannot be changed by any statements within that line. (The `\catcode`, however, of the `\endlinechar` can be changed by statements within the line.)

4.2 Expanding control sequences

In messages to a TEX user it's frequently necessary to refer to control sequences or characters that have special category codes. This can sometimes be problematic because of the expansion that is done in the argument of a `\message` or `\write` command. For example, the line

```
\message{Beware of \footnote in a \message!}
```

will typically generate a hundred or so error messages when TEX reaches `\footnote`. (Although PLAIN TEX, $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX, LATEX, and other macro packages define `\footnote` differently, all the definitions are equally explosive inside a `\message`.) And the line

```
\message{Beware of \endinput in a \message!}
```

will cause the current input file to terminate immediately! (`\endinput` is an *expandable* control sequence, for reasons that are too technical to be worth discussing here.)

Thus to talk about an expandable control sequence in a message, you must do something to inhibit the expansion. Ordinarily you apply `\string` or `\noexpand` to the control sequence; or you could put it into a token register and use `\the`⟨*token register*⟩ in the message. Nonexpandable control sequences can be printed in a message without special protection, except that, if you do nothing to avoid it, you will always get an extra space after a control word, even in some cases where it is undesirable, as when the next thing is punctuation. For example, the message

```
\message{Enter desired value for \hangindent: }
```

will print on screen with a space before the colon:

```
Enter desired value for \hangindent :
```

Table C shows what happens to various sorts of things in a `\message` or `\write` argument, as well as various methods for suppressing expansion.

4.3 Collapsing spaces

If you want to print on screen a menu or similar message consisting of multiple columns nicely arranged, you have to deal with the fact that TEX normally condenses multiple spaces and tab characters to a single space. The easiest way to handle this difficulty is to change the catcode of the space character to, say, 12 before reading the argument of a `\message` or `\write` command.

4.4 Special characters

The space character is but one example of a larger class: 'special' characters, that is, ones that don't have category 11 or 12. An obvious question to ask is, "What other special characters are difficult to use in a message?" Table C shows how a few special characters are affected by the expansion process in a message: an `&` (category 4) passes through unharmed, a `#` (category 6) gets doubled, and a `~` (category 13) gets expanded.[10] Table D is a complete list of the various categories of characters, along with ways to produce those characters that cannot simply be used as is. A few categories deserve more extensive comment.

[9] Although the `^^J` combination will be resolved to a single character by TEX, it is three characters when writing it and when sending it through mail. Assuming, that is, that you use TEX's double caret notation and don't try to insert a `^^J` byte directly (not a good idea, because of system-dependent interpretation of `^^J`).

[10] To be more precise, an active character like `~` will be treated like a control sequence; it will be expanded if expandable, otherwise it will be printed as is. After an assignment such as `\newcount ~`, or `\let ~ = \relax`, the `~` is not expandable.

Table C: Expansion of \message and \write arguments

This input	Produces this on screen
\message{E}	E
\message{&}	&
\message{#}	##
\message{[\relax]}	[\relax]
\message{[\string\relax]}	[\relax]
\message{[\empty]}	[]
\message{[\noexpand\empty]}	[\empty]
\message{[\string\empty]}	[\empty]
\message{[\space]}	[ ]
\message{[ ]}	[ ]
\def\spaces{\space\space\space\space	
\space\space\space\space}	
\message{[\spaces]}	[ ]
\message{[\romannumeral 37]}	[xxxvii]
\message{[\uppercase{a}]}	[\uppercase {a}]
\message{[~]}	[\penalty \@M \ ]
\message{[\ifnum\time<600 Too early for me	
\else Let's go\fi]}	[Too early for me]

Category 0—Escape Character It's normally not a problem to print an escape character because it usually occurs as part of a control sequence, which can be printed using \string (and even that may not be necessary if the control sequence is nonexpandable). Even when the escape character is not, logically speaking, part of a control sequence, it can be sent in a message by letting it combine from TEX's point of view with the following character(s). For example, to send the message `Commands in TeX normally begin with a '\' character`, the backslash doesn't need to be treated as an isolated character; combined with the following apostrophe, it forms a control symbol to which \string can be applied.

The only time this fails is when the backslash must be sent as the very last character of a message. Although this case is extremely unlikely, the solution involves a rather useful little macro:

```
\def\xstring{\expandafter\gobble\string}
\def\gobble#1{}% if this is not already defined
```

With this definition,[11] \xstring not only turns a control sequence into a string of characters, it also removes the leading backslash, so that \xstring\\ will produce a single backslash character, as desired. Another solution that involves setting \escapechar temporarily to −1 would also be possible, provided that the remainder of the message doesn't need to use \string in a normal way, with a printable escape character.

```
\begingroup \escapechar=-1
\message{ ...\string\\}%
\endgroup
```

On the other hand, it might be useful to have a category-12 backslash character always available through a macro, not only for messages but for other purposes as well:

```
\edef\backslashchar{\xstring\\}
```

Then \backslashchar could be used in a message instead of \xstring\\. If you needed to use it frequently you would presumably give it a shorter name.

Categories 1 and 2 Characters of category 1 and 2 can be printed without any problem in a message *if they occur in matching pairs*. For these purposes, character codes are irrelevant; $]_1$ and $*_2$ match up as well as $\{_1$ and $\}_2$. If a single, unmatched character of one of these categories must be printed on screen, \xstring can be used with the corresponding control symbol, e.g., \xstring\{ or \xstring\}.

Categories 5, 9, 14, 15 These categories are similar to category 0. Characters of category 0 (escape), category 5 (end-of-line), 9 (ignored), 14 (comment), and 15 (invalid character) cannot enter a token list [*The TEXbook*, Exercise 7.3], so that, actually, it doesn't make much sense to ask what happens to them in the argument of a \message or \write command, which do not deal with raw characters from an input stream but with token lists. The question is not how to print a character token of category 14 in a message (since that is impossible) but, how to produce a category-12 % when the normal catcode of % is 14. The answer is to use \xstring with the corresponding control symbol, e.g., \xstring\%.

[11] Cf. the answer to Exercise 7.10 in *The TEXbook*. The implicit assumption that \escapechar is in the range 0–255 may not be completely reliable.

Table D: Methods for incorporating various categories of characters in a `\message` or `\write` argument

Catcode	Example	Method
0	`\`	Normally handled as part of a control sequence, except at the very end of a message, in which case use `\xstring\\`
1	`{`	`\xstring\{` if unmatched
2	`}`	`\xstring\}` if unmatched
3	`$`	as is
4	`&`	as is
5	`^^M`	`\xstring\^^M` (see the note below)
6	`#`	`\string#` to avoid doubling, or `\xstring\#`
7	`^`	as is (except in rare combinations like `^^>`)
8	`_`	as is
9	`^^@`	`\xstring\^^@` (see the note below)
10	space	as is, except use `\space`'s to produce multiple spaces
11	a	as is
12	/	as is
13	`~`	`\string~`, `\noexpand~`
14	`%`	`\xstring\%`
15	`^^?`	`\xstring\^^?` (see the note below)

Note: Because of the way `\string` operates, something like `\xstring\Γ` will not produce a single character but three category-12 characters, `^ ^ @`. This may normally be what you want, but it won't be satisfactory if the character in question has a special purpose—perhaps to cause a newline, or to print on-screen as an accented letter.

Then again, a better idea might be to pick one character, make it active (probably `~` since it's already active in most macro packages), and define it to produce category-12 characters by their hexadecimal value. With preliminary definitions such as:

```
\escapechar=-1
\def\twelvechar#1#2{\csname hex#1#2\endcsname}
\expandafter\edef\csname hex5c\endcsname{\string\\}
\expandafter\edef\csname hex25\endcsname{\string\%}
\expandafter\edef\csname hex7e\endcsname{\string\~}
...
\escapechar=`\\
```

it would become possible to write, for example,

```
\begingroup
\let~=\twelvechar
\message{Printing backslash ~5e, percent ~25, and tilde ~7e.}
\endgroup
```

and thus send all manner of special characters by substituting a three-character sequence starting with `~`.

4.5 Space after a control word

In §4.2 it was pointed out that an unwanted space may be printed at the end of a control word under some circumstances. It's equally possible that a wanted space at the end of a control word may disappear under other circumstances. For example, it is not uncommon to see macro writers use the combination `\string\controlseq\space` when a control word occurs as an isolated word in the middle of a message; the final `\space` is necessary because a plain space after `\controlseq` would simply disappear according to TEX's normal rules for finding the end of a control sequence name.

The solution to both of these difficulties is easy: use `\string` if you don't want a space after the control word, and use `\noexpand` if you do want a space.[12] Note: The character tokens produced by `\string` can be passed without harm through any number of subsequent steps, including expansion via `\edef` or similar operators, while `\noexpand` only protects an expandable macro through the first expansion step.

4.6 Outer Control Sequences

You can't send an `\outer` control sequence in a `\message` unless you do something to get around the outerness. To illustrate, I present a transcript of TEX's reaction to the following line:

```
\message{Control-L: ^^L}
```

along with various attempts to recover from the resulting error message. (`^^L` is defined as an active character with the 'outer' attribute in both PLAIN TEX and LATEX.)

```
Runaway text?
Control-L:
! Forbidden control sequence found while scanning text of \message
<inserted text>
                }
<to be read again>
                   ^^L
l.149 \message{Control-L: ^^L
                                          }
? 1
Type <return> to proceed, S to scroll future error messages,
R to run without stopping, Q to run quietly,
I to insert something, E to edit your file,
H for help, X to quit.
? h
I suspect you have forgotten a `}', causing me
to read past where you wanted me to stop.
I'll try to recover; but if the error is serious,
you'd better type `E' or `X' now and fix your file.
?
```

Notice that when I tried to type a `1` to delete the offending `\outer` token, I got instead a help message indicating that token deletion is not an option at this point. (The reason behind this lack of token deletion is fairly technical: TEX was in the middle of a procedure called *get_token* when it stumbled over the `^^L` character; but since token deletion itself involves calling *get_token*, allowing token deletion here would mean calling *get_token* from inside itself—something it was not designed for.)

[12] I hadn't noticed the usefulness of `\noexpand` for this purpose until Michael Spivak drew it to my attention, in a conversation at the 1991 TUG meeting in Boston.

4.7 Semi-verbatim alternative

An alternative way of handling message texts, that eliminates the need to remember special methods for various kinds of message elements, is to temporarily change the catcodes of all special characters while reading the argument of a message. With the following definitions:

```
\def\verbwrite{\begingroup
  \def\do##1{\catcode`##1=12}%
  \do\ \dospecials
  \catcode\endlinechar=12
  \newlinechar=\endlinechar
  \verbcontinue}
\begingroup \lccode`\/=`\\
\lowercase{\endgroup
\def\verbcontinue##1/}endverbwrite{%
  \immediate\write16{#1}\endgroup}
```

you could send messages like

```
\verbwrite !#$%$%#^%&*~@^^"<?:}{>|+
_^+\footnote_|)\\90\
\bye ^^L \endinput
\endverbwrite
```

without regard to the contents. The main limitation of this approach is that in order for the handling of the special characters to work, `\verbwrite` has to be executed directly in a file; it cannot be embedded in a macro. Furthermore, the message text is unalterable: it cannot contain a context-dependent part, as in a message to display the current font name on screen:

```
\message{Current font is \fontname\font.}
```

This message could not be generated with `\verbwrite` because `\verbwrite` would not expand `\fontname`.

4.8 Presenting information in the best possible form

Example: In reporting a dimension to the user, it is usually desirable to report the value rounded to tenths or hundredths, in units that are convenient for the user: points for a font size or line spacing value; centimeters, picas, or inches for the height or width of a page or of an included figure.

The file `cnvunits.tex` gives some examples of what is possible in this vein, including conversions from points to picas, inches, and centimeters. The conversions from points to other units are the most important ones because when `\the` is applied to a dimension or skip register TEX always reports the value in `pt` units. (Not counting `\muskip` registers, where the reported unit is `mu`.)

```
% Copyright 1994 Michael John Downes
% Copyright 2013 TeX Users Group
% This file is part of the dialogl package, released under the LPPL;
% see dialogl.ins for details.

\newdimen\zdim \newdimen\tempdim \newdimen\tempdima

% Conversion factors:
%
%       According    Scaled   Rational       Prime
% Unit  to TeX       Points   Form           Factorization
%-----------------------------------------------------------------
% 1 sp  0.00002 pt   1        1/65536 pt     1 / 2^16
% 1 mm  2.84526 pt   186467   7227/2540 pt   3*3*11*73 / 2*2*5*127
% 1 cm  28.45274 pt  1864679  7227/254 pt    3*3*11*73 / 2*127
% 1 pt  1.0 pt       65536    100/7227 in    2*2*5*5 / 3*3*11*73
% 1 pc  12.0 pt      786432   12/1 pt
% 1 dd  1.07 pt      70124    1238/1157 pt   2*619 / 13*89
% 1 cc  12.8401 pt   841489   14856/1157 pt  2*2*2*3*619 / 13*89
% 1 bp  1.00374 pt   65781    803/800 pt     11*73 / 2*2*2*2*2*5*5
%                             [1/72 in]
% 1 in  72.2699 pt   4736286  7227/100 pt    3*3*11*73 / 2*2*5*5
%       [2.54 cm]             [254/100 cm]

\def\points#1#2#3{\tempdim#2\relax
  \edef#3{\csname cnvunits#1\expandafter\endcsname\the\tempdim}%
}

\def\inches#1#2#3{%
  \tempdim=#2\relax
  \tempdima=\ifdim\tempdim<\zdim -\fi\tempdim % absolute value
  \roundup\tempdima{#1}{in}%
%    In the interest of maximum accuracy we push \tempdima as near
%    to \maxdimen as possible before dividing, using the prime
%    factorization of the fraction 7227/100 which is the
%    points/inches conversion factor.
  \ifdim\tempdima<.01\maxdimen
    \multiply\tempdima 100 \divide\tempdima 7227
  \else
    \ifdim\tempdima<.1\maxdimen
      \multiply\tempdima 10 \divide\tempdima 11
      \multiply\tempdima 10 \divide\tempdima 657
    \else
      \divide\tempdima 9 \multiply\tempdima 5
      \divide\tempdima 803 \multiply\tempdima 20
    \fi
  \fi
  \tempdim=\ifdim\tempdim<\zdim -\fi \tempdima
  \edef#3{%
    \csname cnvunits#1\expandafter\endcsname\the\tempdim}%
}

%    Function \roundup for rounding upward. #1 must be a dimension
%    register. If it holds a negative value it will be rounded
%    'outward' away from zero rather than 'upward' toward zero. #3
%    is a TeX units string such as "pt" or "in". If #2 = 0 then
%    this will round up to the nearest tenth; if #2 = 00, nearest
%    hundredth; and so forth (up to 5 zeros). If #2 is empty then
%    full accuracy up to TeX's limits will be used.
%
%    The rounded result will be returned in the dimension register
%    #1.

\def\roundup#1#2#3{%
  \if .#2.\else
    \begingroup
      \ifdim#1>\zdim
        \advance#1-\maxdimen \advance#1.#25#3\relax
      \fi
      \ifdim#1<\zdim
    \endgroup
        \advance#1.#25#3
      \else
    \endgroup
      \fi
  \fi
}

\begingroup
\catcode`\P=12 \catcode`\T=12
\lowercase{%
\expandafter\gdef\csname cnvunits\endcsname#1PT{#1}
\expandafter\gdef\csname cnvunits0\endcsname#1.#2PT{%
  #1.\takeone#20\takeone}
\expandafter\gdef\csname cnvunits00\endcsname#1.#2PT{%
```

```
  #1.\taketwo#200\taketwo}
\expandafter\gdef\csname cnvunits000\endcsname#1.#2PT{%
  #1.\takethree#2000\takethree}
\expandafter\gdef\csname cnvunits0000\endcsname#1.#2PT{%
  #1.\takefour#20000\takefour}
\expandafter\gdef\csname cnvunits00000\endcsname#1.#2PT{%
  #1.\takefive#200000\takefive}
}%
\endgroup

\def\takeone#1#2\takeone{#1}
\def\taketwo#1#2#3\taketwo{#1#2}
\def\takethree#1#2#3#4\takethree{#1#2#3}
\def\takefour#1#2#3#4#5\takefour{#1#2#3#4}
\def\takefive#1#2#3#4#5#6\takefive{#1#2#3#4#5}

\def\showinches#1{\inches{00}{#1}\converted
  \immediate\write16{%
    #1 = (after conversion) \converted\space inches}}

\showinches{0in} \showinches{1in} \showinches{2.0in}
\showinches{2.2in} \showinches{8.5in} \showinches{1pc}
\showinches{6pc} \showinches{1cm} \showinches{1mm}
\showinches{1bp} \showinches{72bp} \showinches{1cc}
\showinches{1dd} \showinches{72dd} \showinches{5000pt}
\showinches{-5000pt} \showinches{\maxdimen}
\showinches{-\maxdimen} \showinches{.999\maxdimen}
\showinches{1pt} \showinches{.01pt}

\endinput

%    From the TeX log:

0in = (after conversion) 0.00 inches
1in = (after conversion) 1.00 inches
2.0in = (after conversion) 2.00 inches
2.2in = (after conversion) 2.20 inches
8.5in = (after conversion) 8.50 inches
1pc = (after conversion) 0.17 inches
6pc = (after conversion) 1.00 inches
1cm = (after conversion) 0.39 inches
1mm = (after conversion) 0.04 inches
1bp = (after conversion) 0.01 inches
72bp = (after conversion) 1.00 inches
1cc = (after conversion) 0.18 inches
1dd = (after conversion) 0.01 inches
72dd = (after conversion) 1.07 inches
5000pt = (after conversion) 69.18 inches
-5000pt = (after conversion) -69.18 inches
\maxdimen  = (after conversion) 226.70 inches
-\maxdimen  = (after conversion) -226.70 inches
.999\maxdimen  = (after conversion) 226.48 inches
1pt = (after conversion) 0.01 inches
.01pt = (after conversion) 0.00 inches
```

Another example: if you want to report the `\mathcode` of a particular character to the user, `\number\mathcode‘\x` or `\the\mathcode‘\x` aren't too great, because they produce a decimal number, when it would be more convenient to get a hexadecimal number filled out to four digits, so that the class, math family, and font position information can be read off directly. Extending some ideas from `testfont.tex` [Knuth, 1986c], we can write a quite friendly `\reportmathcode` function:

```
%    When \meaning is applied to a \mathchar, it produces
%    \mathchar"<digits> where <digits> are 1 to 4 hexadecimal digits.
%    The function \gethex strips off the prefix and leaves only the
%    digits.
\def\gethex#1"{}
%    The function \reportmathcode takes a character or control symbol
%    argument and reports the associated mathcode in hexadecimal
%    form, filling out to four digits with leading zeros, if
%    necessary.
\def\reportmathcode#1{%
  \begingroup
    \mathchardef\temp=\mathcode‘#1 \relax
    \edef\temp{\expandafter\gethex\meaning\temp}%
    \count@="\temp\relax
    \edef\temp{%
%    Fill in leading zeros
      \ifnum\count@<"1000 0%
      \ifnum\count@<"100 0%
      \ifnum\count@<"10 0\fi\fi\fi
      \temp}%
    \message{The mathcode of \string#1 is: "\temp}%
  \endgroup}
```

Example 2: The `\printoptions` command of $\mathcal{AMS}$-TEX version 1.1. `\W@` is the $\mathcal{AMS}$-TEX abbreviation for `\immediate\write16`

```
\def\S@{S } \def\G@{G } \def\P@{P }
\newif\ifbadans@
\def\printoptions{\W@{Do you want S(yntax check),
  G(alleys) or P(ages)?^^JType S, G or P, follow by <return>: }%
 \loop \read\m@ne to\ans@
   \xdef\next@{\def\noexpand\Ans@{\ans@}}%
   \uppercase\expandafter{\next@}%
   \ifx\Ans@\S@\badans@false\syntax\else
   \ifx\Ans@\G@\badans@false\galleys\else
   \ifx\Ans@\P@\badans@false\else
   \badans@true\fi\fi\fi
   \ifbadans@\W@{Type S, G or P, follow by <return>: }%
 \repeat}
```

5 Stumbling blocks in the use of `\read`

5.1 An example: $\mathcal{AMS}$-TEX's `\printoptions` command

Consider the `\printoptions` command of $\mathcal{AMS}$-TEX 1.1 (Example 2): The definition of this command shows one way of dealing with the extra space at the end of a macro created using `\read`: Define some macros consisting of the expected answers, with the extra space included, and then use `\ifx` to compare them to the user's response. It also shows how to uppercase the user's response so that lower- and uppercase responses will be treated identically. This is the second method given in the answer to *The TEXbook*'s Exercise 20.19. One more noteworthy feature of `\printoptions` is that it runs a loop that doesn't quit until the user gives an acceptable answer.

In `\printoptions` since `\W@` is defined to be `\immediate\write16`, and the `\write` command always starts a new line after its message text, we can see that the reply typed by the user will appear on the next line instead of immediately after the colon. This brings up the question: what if we want the user's reply to appear on the same line?

One way to do this is to use `\message` to send the last line of the prompt message, and use `\write` to send the previous lines. For example:

```
\W@{Do you want S(yntax check), G(alleys) or P(ages)?}%
\message{Type S, G or P, follow by <return>: }%
```

This idea is used in the LaTeX option `checknum.sty` [Hamilton Kelly, 1991]. An alternative would be to put the whole prompt in a single `\message` with embedded newline characters (as long as you are careful to select a character for `\newlinechar` that is not needed in the text of the message).

5.2 ^^M at the end of a line

In `\printoptions` separate macros `\S@`, `\G@`, and `\P@` are defined for each legitimate response. If the menu becomes more extensive, this technique is rather wasteful of hash size, main memory, and other useful commodities. The problem here is that the `^^M` character at the end of the user's response is included by `\read` in the macro being read. Under normal conditions `^^M` is converted to a space; however, another possibility—if the user just enters RETURN without typing any response—is that the `^^M` will produce a `\par` token (following the general rule that an empty line is equivalent to `\par`). The best approach is to prevent the `^^M` character from getting into the read macro in the first place. This can be done in two ways: setting the catcode of `^^M` to 9 ("ignore"), or setting `\endlinechar` to −1.

Unfortunately, this immediately raises another difficulty: we want to keep the catcode change or `\endlinechar` change local so that it will affect only the `\read`. This could be accomplished by saving the current catcode or `\endlinechar` (just in case) and restoring it after the `\read` is done, but it's simpler to enclose the `\read` in a group:

```
\begingroup
\endlinechar=-1
\global\read16 to\answer
\endgroup
```

Here the `\global` prefix makes `\answer` retain its definition beyond the `\endgroup`.

With this modification the tests done by `\printoptions` could be simplified to

```
\if\Ans@ S ...\else
\if\Ans@ G ...\else
\if\Ans@ P ...\else
...
```

which renders the macros `\S@`, `\G@`, `\P@` unnecessary.

On the other hand, we have advanced to some splendid new complications: `\Ans@` might now be completely empty, if the user just pressed the RETURN key, and an empty `\Ans@` would bollix up the `\if` tests. This case is easy to handle, though: add an extra branch `\ifx\Ans@\empty...` at the beginning. We have the opposite problem if the user types more than one letter: on the true branch (the 'none-of-the-above' branch, unless the user's first two letters happen to be identical), the extra characters could potentially cause spurious typesetting activity. As it happens, we can kill two birds with one stone, as we'll see in §5.4.

5.3 Uppercasing input

Next let's look at the procedure used by `\printoptions` for uppercasing the user's reply: after reading `\ans@`, `\xdef` and `\uppercase` are applied to it as follows:

```
\xdef\next@{\def\noexpand\Ans@{\ans@}}%
\uppercase\expandafter{\next@}%
```

A more economical version of the same technique would be:

```
\xdef\ans@{\uppercase{%
  \gdef\noexpand\ans@{\ans@}}}%
\ans@
```

If `\ans@` contains `s` to begin with, then after the `\xdef` has been completed, the definition of `\ans@` is `\uppercase{\def\ans@{s}}`. Then calling `\ans@` causes it to redefine itself, but not before the tokens in the argument of `\uppercase` are suitably uppercased.[13] With this change, the auxiliary macro `\Ans@` is no longer needed.

To simplify the structure of macros using this uppercasing process, it could be embodied in a dedicated function of its own:

```
\def\uppermac#1{\xdef#1{\uppercase{\gdef\noexpand#1{#1}}}%
  #1}
```

A nonglobal alternative may sometimes be desirable, however. Also the full expansion may not be wanted in some cases when the contents of the macro being uppercased are 'fragile'. An alternative that is safer with respect to expansion:

```
\def\uppermac#1{%
  \uppercase\expandafter\expandafter\expandafter{%
    \expandafter\toks@\expandafter{#1}}%
  \edef#1{\the\toks@}}
```

5.4 Default responses

One last refinement in `\printoptions` would be to provide a default response if the user's response is empty (that is, the user only hit the RETURN key). One method involves an auxiliary macro like the LaTeX macro `\@car`:

```
\def\@car#1#2\@nil{#1}
```

A more descriptive name (for those whose knowledge of Lisp is nil) would be `\firsttoken`:[14]

```
\def\firsttoken#1#2@{#1}
```

Then, if we want `\ans@` to be given a default value of P when it comes back empty from the user, we do this:

```
\xdef\ans@{\expandafter\firsttoken\ans@ P@}
```

At the critical intermediate step, the following cases will arise:

[13] Only the `s` is affected because `\uppercase` operates only on letters, not on control sequences or nonletters. Well, to be more precise: only on characters that have a nonzero `\uccode`; they don't have to have catcode 11.

[14] Using `@` as the ending delimiter is pretty safe if we make sure that it has catcode 11 at the time `\firsttoken` is defined and some other catcode at the time of reading user input.

User input	Critical step
s	`\firsttoken s P @`
P	`\firsttoken P P @`
⟨*return*⟩ | `\firsttoken P @` |

This gives exactly what we want.

The application of `\firsttoken` also gives us a nice way around the difficulty mentioned earlier if the user types more than one character. The case

synt	`\firsttoken s y n t P @`

will produce the same result as the first case above, because everything after the first `s`, up to the category-11 `@` character, will be discarded.

5.5 A new \printoptions

By noticing that the `\xdef`'s used in the `\firsttoken` step and the `\uppercase` step can be combined, and putting together everything discussed so far, we come up with a new, improved version of `\printoptions`:

```
\def\printoptions{%
  \W@{Do you want S(yntax check), G(alleys) or P(ages)?}%
  \message{Type S, G or P, follow by <return>: }%
  \begingroup \endlinechar\m@ne
  \global\read\m@ne to\ans@
  \endgroup
  \xdef\ans@{\uppercase{%
    \def\noexpand\ans@{%
% Default to 'P':
      \expandafter\firsttoken\ans@ P@}%
    }}%
% Execute \ans@ to uppercase itself:
  \ans@
  \if S\ans@ \syntax\else
  \if G\ans@ \galleys\else
  \if P\ans@ % fine, no action needed
  \else \message{Unknown option: \ans@;
     'pages' option will be used}\fi
  \fi\fi
}
```

The loop has been discarded in favor of simply taking the normal default action if the user's reply is unintelligible.

5.6 Matching braces

The `\read` command normally reads only one line, but if the first line does not contain an equal number of left and right braces, TEX will continue to read additional lines until equality is achieved (cf. §3.1). This could be trouble in interactive use of `\read`, if the user doesn't understand what has happened—it's difficult to extricate yourself except by getting the braces right.

On the other hand, if you *want* to enter more than one line at a time, you can do it by entering an opening brace on the first line and the matching closing brace on the last line; this is illustrated in Example 3. In the example there are two things worthy of note: (1) The braces appear in the replacement text of the macro `\name`; this may be undesirable, depending on the intended use of the information. (2) There's no space between Frank and Henry in `\name`.

The loss of the space has two causes. First, `\endlinechar` was set to −1 (so that an empty line will not produce a `\par`—see §5.2), and second, the space that was typed at the beginning of the second line of the response didn't register either, following TEX's usual rule of ignoring spaces at the beginning of a line (*The TEXbook*, Chapter 8, double dangerous bends).

5.7 Outer macros

[This section is partly obsolete as of TEX version 3.141; see §3.1.]

If you enter an `\outer` macro in response to a `\read` prompt, TEX will inform you in an error message that it has inserted a closing brace. Unfortunately, this is rather unhelpful, since you will then have the matching brace problem described in the previous section; your answer now contains an unmatched right brace, and if you don't type 'x' at the question-mark prompt to exit, you could get stuck. Fortunately, it is unlikely for anyone to ever enter an outer macro in response to a prompt, since in PLAIN TEX the set of such macros is small and used relatively infrequently, and in LATEX there are almost no `\outer` macros at all; accidental typing of a `^^L` character (which is active and outer, in PLAIN TEX and LATEX) is perhaps the least unlikely possibility.

Some observations:

—If the user checks the help message and stops to ponder the situation, they have the opportunity, at least, to realize that E or X to exit is indeed the wise choice. They're not *really* stuck unless they carelessly try to continue.

—If the user doesn't avail him/herself of the E or X option, just about anything else that they try will be ineffective. On some systems even the interrupt key won't help here; that leaves essentially two ways out: match up the closing brace, or type another outer thing to get back to the `?` prompt and the associated error recovery opportunity.

5.8 Catcodes

The treatment of a user's response depends on the use to which it will be put. LATEX's `\typein` command is designed to take the response and execute it, and therefore reads the user's response using normal category codes. On the other hand, `testfont.tex` [Knuth, 1986c] changes the category codes of the special characters to 12 when reading a user response, because the response will not be executed but will be processed as simple character data. This approach is probably the better one for most applications, since it avoids the possibility of problems with things like mismatched braces or outer control sequences, and since the `\read` command is used more often to read strings of ordinary characters than to read executable control sequences.

Example 3: Reading multiple lines with a single \read command

First, the input file:

```
\begingroup \newlinechar=‘\& \endlinechar=-1
\message{&Please enter your name: }\global\read-1 to\name
\message{&And your Social Security number: }\global\read-1 to\ssno
\endgroup
\show\name \show\ssno
...
```

Now the log file, including the responses (a RETURN was typed after Frank):

```
Please enter your name: {Joe Bob Willie Clark Mark Raphael Ferguson Frank
 Henry James Percival Emerson Elmo Davenport, Jr.}

And your Social Security number: 360-60-6000
> \name=macro:
->{Joe Bob Willie Clark Mark Raphael Ferguson FrankHenry James Percival Emerson
 Elmo Davenport, Jr.}.
l.7 \show\name
               \show\ssno
?
> \ssno=macro:
->360-60-6000.
l.7 \show\name \show\ssno

?
```

5.9 Latex.tex: \typeout and \typein

An interesting aspect of the \typeout and \typein commands in LaTeX is that they aren't private control sequences (with @ characters in their names); they are available for use in ordinary document files. One of the uses suggested in the LaTeX manual is for entering an \includeonly command interactively each time a multipart document is processed. More commonly, however, \typeout and \typein are used internally in documentstyle files, or in special applications such as docstrip.tex [Mittelbach, 1991] where the distinction between private and public control sequences is irrelevant.

The purpose of \typein is (a) to print a message on screen, and (b) to read a response (one line) from the user, either into the internal macro \@typein, or into a macro chosen by the macro writer. If \@typein is used to receive the response, it will be executed as \typein's final action. Otherwise the response will be stored in the designated alternate macro, without execution. (Cf. the LaTeX manual, §4.6.)

I found the definition of \typein in latex.tex more difficult to understand than almost anything else of comparable length that I have looked at. All the complications in the definition serve two goals: (1) If the user simply presses the RETURN key, the resulting \par token needs to be discarded, leaving the macro that holds the user reply empty; and (2) if the user reply is not empty, it will usually, but not always, contain a final space which needs to be trimmed off. The fact that the final space might be missing is the crucial problem.

A simpler version of \@xtypein could be written using a temporary deassignment of \endlinechar:

```
\def\@xtypein[#1]#2{\typeout{#2}\let\@typein\relax
  \begingroup \endlinechar\m@ne \global\read\z@ to#1\endgroup
  \@typein}
```

This solves both the \par problem and the trailing space problem.

However, much of the initial development of LaTeX took place in 1982 and 1983, before the ultimate release of TeX82, version 1.0 (officially: December 3, 1983), and in old TeX there was no access to \endlinechar. (In fact many features of TeX82 were added by Knuth in response to reports from Lamport about various limitations of the language that he ran into in the course of LaTeX's development.) If Lamport noticed later that \endlinechar could be applied in \@xtypein, he probably invoked the principle 'If it ain't broke, don't fix it' and left it alone.

5.10 Docstrip.tex: \typeout, \typein, progress reports

As an example of the use of \typeout and \typein, consider docstrip.tex [Mittelbach, 1991]. This is a LaTeX utility used for processing a documented macro file to remove comments (the stripped-down version of a large macro file will load significantly faster at run-time, at least on less powerful computers). The use of \typeout in docstrip.tex is mainly a convenience, to avoid the more cumbersome phrase \immediate\write16, but \typein has a more significant advantage—it takes care of removing a space at the end of the user response, if present.

In the following fragment from docstrip.tex, the user is informed that an auxiliary file named docstrip.cmd has been detected, and is asked whether it should be used. The

Example 4: Outer control sequence choking a \read operation. Compare this to the behavior of ^^L inside \message as discussed in §4.6.
Input file:

```
\read16 to\x  \show\x
\end
```

Log file:

```
\x=^^L
Runaway definition?
->
! Forbidden control sequence found while scanning definition of \x
<inserted text>
                }
<read *> ^^L

l.1 \read16 to\x

? h
I suspect you have forgotten a `}', causing me
to read past where you wanted me to stop.
I'll try to recover; but if the error is serious,
you'd better type `E' or `X' now and fix your file.

? 1
Type <return> to proceed, S to scroll future error messages,
R to run without stopping, Q to run quietly,
I to insert something, E to edit your file,
H for help, X to quit.
?
a
\
#
{
> \x=macro:
-> } \par a \^^M## { .
l.2 \show\x

?
```

response is read into the macro \answer.

```
\typeout{*************************************************}
\typeout{* Batch file docstrip.cmd found. Use it ? (y/n) *}
\typein[\answer]
    {*************************************************}
\def\@tempa{y}%
\ifx\@tempa\answer \process@batchfile \fi
```

An analysis of the technique in this fragment brings out a few noteworthy observations.

- In the prompt (line 2) the acceptable responses are listed: y or n.
- Since \typein doesn't change the case of the user's response, and there's no code here in docstrip.tex to change case, we can deduce that a response of Y will be treated as a No.
- If the user response is yes, then \answer and \@tempa will have different definitions, and the \ifx test will be false.
- If the user is expected to enter only a single letter, we might ask, why is an \ifx test necessary? Why not just use \if y\answer\process@batchfile and omit the extra step \def\@tempa{y}? However, this would invite trouble if the user pressed RETURN without typing anything. Then \answer would be empty (because of the special handling with \@defpar in \typein) and before completing the evaluation of the \if, TEX would continue beyond \answer to expand \process@batchfile in search of a character to compare with the y. Although this would give a correct result for the \if test, because the first nonexpandable token in the definition of \process@batchfile is not a y, the trouble is that the token will be absorbed by the \if operation, which will throw a wrench into subsequent processing.

Note. The version of docstrip.tex from which I took these examples was an early, unperfected version with an explicit disclaimer from the author that it was a quick hack, and should not be considered an example of his best macro writing. I discuss the early version here, however, precisely because it illustrates so well some of the typical difficulties in writing TEX dialog.

An interesting feature of docstrip is the ability to see on screen the progress being made in processing the current file. If this feature is turned on (with a \showprogress command), then docstrip prints on screen a percent sign for each comment removed and a period for each line of code that is kept. This kind of visual feedback is not just frippery and ornamentation, bells and whistles. A typical user begins to get nervous if a running program allows several seconds to pass by without any observable change on screen. On a fast system, it's straightforward to deduce that such a pause means something has gone wrong, whereupon the user can press an interrupt key to investigate the problem without wasting time. But on a slow system the poor user faces the dilemma: Abort unnecessarily, and a restart will be required, with all the time consumed so far wasted; or continue waiting when the program might simply be spinning its wheels and accomplishing nothing. So when a task might take more than, say, five seconds, it's good programming practice to give some sort of progress reports.

The progress reports of docstrip use \message which, as discussed earlier, adds a space between consecutive messages if the second message is not too long to fit on the current line. So what you see on screen is something like

```
% % % % % % % % % % . . . % % . . . . . % % % . . % . % % . % . %
. . . % % . % . % % % % % % % % % . . % . % % . % % % % . . . % .
. . .
```

produced by repeated \message{%} or \message{.}. Recalling the newline behavior of \immediate\write, picture what would happen if it were used instead of \message to see why \message is necessary for this application.

5.11 emTEX 8-bit output

One interesting feature of emTEX [Mattes, 1992] is the 8-bit output option. On PCs, the characters between 176 and 223

are box-drawing characters; with the 8-bit output option of emTEX, putting these characters into a `\message` or `\write` command allows you to draw some fairly elaborate boxes on screen, for embellishing menus and other bits of dialog. I have only experimented with this a very little.

5.12 User Help

There is a good deal of room for improvement in the amount and kind of help information available to the user from within TEX. Help information provided externally through general help facilities such as Unix man pages is well and good, but any help system that's not TEX-based has one disadvantage: lack of portability across the whole spectrum of computer systems that can run TEX.

- Any program should have an announcement near the beginning of how to quit without destructive side effects; for TEX this means, among other things, that each version of the TEX program should have in its opening message instructions on how to break out of an infinite loop or in general how to interrupt TEX before it has finished its current run. In *Textures* this requirement is satisfied by the 'Pause' button, always visible. In DOS versions of TEX the interrupt key is normally the Control-Break or Control-C key (depending partly on the particular implementation of TEX); in VAX/VMS it is Control-C or Control-Y, with the latter reserved for emergency use only, since it will leave you without a log file for reference.
- Response to the prompt

  ```
  I can't find file xxx.  Please type another
  input file name:
  ```

  The novice user should be able to type the reasonable guesses `help` and `h` and `?`; the easiest way to do this would be to put files `help.tex`, `h.tex` or `?.tex` in the TEX inputs directory/folder/area. (Except that many OS's don't allow `?.tex` as a file name). Already many systems have a file `null.tex` or `nul.tex` to allow you to abort reasonably gracefully if you know enough to enter `null` or `nul` in response to the prompt. Very few TEX users, however, will ever think of entering `nul` without reading about it in the documentation or hearing about it from a more experienced user.
- Some of TEX's *built-in* error/help messages are specific to `plain.tex`. Some big macro packages such as LATEX might prefer to change some of the wording at least. For example, there are references in some of TEX's compiled-in help messages to things like `\def` and `\eqalign` that are documented nowhere in the LATEX book. And if you press RETURN at a `*` prompt after getting into TEX's interactive mode (intentionally or accidentally), TEX urges you to enter a command or type `\end`—the latter being worse than useless in LATEX (where `\stop` or `\end{document}` are what is required): after typing `\end` and RETURN, nothing happens because LATEX is waiting for the argument of the `\end` command.
- Long help and error messages use string pool and main memory. Storing them in external files would provide more space (at the cost of slower access; but of course, once you get an error message, processing speed is scarcely relevant any longer).

6 Summary

6.1 Sending messages

Recommendations: Until versions of TEX earlier than 3.141 are sufficiently phased out, you had better use `\immediate \write` rather than `\message` for generic message-sending macros, so that all "printable" characters remain available for use in the message text. Use `\message` instead of `\immediate\write` for producing a prompt if you want the user's response to appear on the same line. Uncatcode all special characters while constructing the text of a message, if the message text is completely invariant between one use and the next. Use `\string` if you don't want a space after a control word, and use `\noexpand` if you do want a space.

6.2 Reading user input

Recommendations: Set the catcode of `\endlinechar` temporarily to 9 while reading a response, to avoid getting an extra space at the end from the `^^M`. Uncatcode all special characters, especially opening and closing braces. If it is not uncatcoded, remove the outerness from `^^L`, at least while reading a user response, and similarly, if the backslash is not uncatcoded and you want to be supremely cautious, remove the outerness from any other outer control sequences (e.g., `\newif`) if they might reasonably, or even unreasonably, turn up in a user's response.

References

[Cowan, 1987] Cowan, Ray. `tables.sty`. 1987. This is derived from `tables.tex`; I found it at `sun.soe.clarkson.edu`, directory: `pub/tex/latex-style`.

[Greene, 1989] Greene, Andrew Marc. "TEXreation—Playing games with TEX's mind." *TUGboat* 10(4), pages 691–705, 1989. Includes a listing of `animals.tex`.

[Greene, 1990] Greene, Andrew Marc. "BaSiX: An interpreter written in TEX." *TUGboat* 11(3), pages 381–392, 1990.

[Hamilton Kelly, 1991] Hamilton Kelly, Brian. `checknum.sty`. *UKTEX* 91(1), 4 January 1991.

[Knuth, 1986b] Knuth, Donald E. *TEX: The program*. Reading, Mass.: Addison-Wesley, 1986.

[Knuth, 1986c] Knuth, Donald E. `testfont.tex`. *The METAFONTbook*, Appendix H, section 4. Reading, Mass.: Addison-Wesley, 1986. This file is included in all standard distributions of METAFONT.

[Lamport, 1985] Lamport, Leslie. `latex.tex`. Version 2.09 (1985–1992). Main source file for LATEX, included with any standard distribution.

[Mattes, 1992] Mattes, Eberhard. emTEX. Version 3.1415. A comprehensive suite of programs including TEX, METAFONT,

printer drivers, previewers, BibTEX, ... Available by anonymous ftp from `niord.shsu.edu` (USA) or `ftp.uni-stuttgart.de` (Europe) and other fine archives.

[Mittelbach, 1991] Mittelbach, Frank. `docstrip.tex`. Version 1.1l, 1991. This file is part of the `multicol` package available by anonymous FTP from many archives, including `ftp.uni-stuttgart.de` and `ymir.claremont.edu`.

Appendix A Basix.tex

Another effort by Andrew Marc Greene, with clear relevance to the subject of dialog in TEX, is his prototype Basic interpreter described in [Greene, 1990]. I had planned to give here a closer study of the dialog concepts used by `basix.tex` but it seems I will not have enough time.

Appendix B Tables.tex

The file `tables.tex` [Cowan, 1987] provides table macros with the unique property that a preamble line specifying the format of each row is not required; the format is determined automatically by an analysis of the table contents. The dialog part consists of a message such as

```
[Nrows=9, Ncols=2]
```

that is printed on screen for each table. This provides confirmation at run-time for the user of the general structure of each table. In the worst case, if the number of rows or columns is wildly wrong, the user can press the interrupt key and go fix up the table before trying again.

Appendix C Fontmenu.tex

The file `fontmenu.tex` demonstrates a crude menu system based on the token deletion option after an error message. There are five tokens `\ComputerModern`, `\Garamond`, etc., corresponding to the five font choices. They are so defined and arranged that if the user enters, say, 2 to select Garamond fonts, then the deletion of two tokens will leave the `\Garamond` token showing on screen (as the last token deleted by the user), and then the next token (`\Helvetica`) will define the font base to be 'Garamond' and gobble the remainder of the list. The effect of this arrangement is that the user sees the '`\Garamond`' on screen as a confirmation of their selection after they enter the number and before they press the RETURN key a second time.

```
% Copyright 1994 Michael John Downes
% Copyright 2013 TeX Users Group
% This file is part of the dialogl package, released under the LPPL;
% see dialogl.ins for details.

\def\ComputerModern{\gdef\fontbase{Times}\gobble}
\def\Garamond{\gdef\fontbase{Computer Modern}\gobble}
\def\Helvetica{\gdef\fontbase{Garamond}\gobble}
\def\Malibu{\gdef\fontbase{Helvetica}\gobble}
\def\Times{\gdef\fontbase{Malibu}\gobble}

\def\gobble#1\endgobble{}

%    Make sure we have a reasonable \newlinechar
\newlinechar=`\^^J \catcode\newlinechar=12

\begingroup
\catcode`\<=1 \catcode`\ =2\relax
\gdef\menustart{\errmessage<%
..................................... }%
\endlinechar\newlinechar\catcode`\ =12\relax
\gdef\menutext{
 ....................................
 ....................................

  Select the font base you wish to use:

  [1] Computer Modern        [4] Malibu
  [2] Garamond               [5] Times
  [3] Helvetica

  (Default: Times)}%
\endgroup% percent here to avoid extra \newlinechar

\begingroup
\immediate\write16{\menutext}
%    If the user accidentally types 33 instead of 3 they will get past
%    all the legitimate menu choices. So to keep that from causing
%    trouble, we throw in a bunch of ~ characters below to perform
%    error recovery. The standard maximum number of tokens that TeX
%    will delete at one time is 99.
\catcode`\~=\active % just to make sure
\def~{\newlinechar`\^^J% restore normal value
  \message{%
Whoops! Well, you got Times as your font base, I think}%
  \gobble}
%    Minimize unwanted error context (note: setting
%    this to -1 doesn't suppress any more information)
\errorcontextlines 0
\newlinechar=`\! % to hide the automatic ! from \errmessage
\gdef\fontbase{Times}
%    Inside the next group we make the space character
%    active so that we can  use it to call \menustart,
%    and then we use \expandafter so that the first
%    space on the next line gets that catcode before
%    the \endgroup makes it revert to normal. All this
%    so that the user does not see `\menustart' on
%    screen, only a space.
\begingroup
\catcode`\ =\active\let =\menustart\expandafter\endgroup%
 % Enter a number (1...5) and press Return TWICE
\ComputerModern% Press Return to continue
\Garamond% Press Return to continue
\Helvetica% Press Return to continue
\Malibu% Press Return to continue
\Times% Press Return to continue
\gobble%
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~%
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~%
\endgobble\endgroup%
\show\fontbase
\end
```

And here is screen output of a typical run through `fontmenu.tex`:

```
This is emTeX, Version 3.0 [3a] (preloaded format=plain 93.9.20)
  30 OCT 1993 23:41
**&plain fontmenu
(fontmenu.tex

 ....................................
 ....................................

  Select the font base you wish to use:

  [1] Computer Modern        [4] Malibu
  [2] Garamond               [5] Times
```

```
  [3] Helvetica

  (Default: Times)

 ....................................
 ....................................

l.56
      % Enter a number (1...5) and press Return TWICE
? 4
l.60 \Malibu
            % Press Return to continue
?
> \fontbase=macro:
->Malibu.
l.66 \show\fontbase

? x
No pages of output.
```