Dialog with TEX

Michael J. Downes
49 Weeks Street
North Smithfield, RI 02895
mjd@math.ams.com

Introduction

On the face of it, of course, 'dialog with TEX' doesn't make much sense, because TEX isn't a person that can carry on a conversation. The truth is that a team of real persons, Knuth and the macro writer (or writers), have tried to anticipate the user's side of the conversation and prepare good answers in advance. It's these packaged answers, relayed to the user through TEX, that form the other side of the conversation.

Yet when we deal with a computer program such as TEX, our human tendency is to translate the pseudo-conversation, carried on by printed messages on the computer screen, into a more familiar framework: natural language conversations with real people. This is done easily enough by pretending that the program is a sort of genie (or lion, in the case of TEX) that happens to live inside the computer. This pretense is particularly convenient in writing about programs, where it helps cut down on awkward circumlocutions.

TEX can be run in batch mode or interactive mode, but the most frequent way of running TEX might best be called *barely interactive*: you start running TEX in interactive mode and give it a file name to process, whereupon TEX typesets the file and quits, without needing any further input from you — but you hang around anyway, in case an error occurs, because if so then you have to type something in response to the question mark prompt, before TEX can finish processing the file. If you don't expect any errors, however, you could go get a cup of coffee while TEX is running, or in the case of a long document maybe even go home and mow the lawn.

By *dialog with TEX*, then, I don't mean errorless typesetting runs where the presence of the user is immaterial; I mean two-way communication with the active participation of the user, not only in responding when TEX prompts for a response, but also in paying attention to any messages TEX may send, whether they require a response or not. More generally, I'll define *dialog with TEX*, for the purposes of this discussion, as *the communication of interesting information, in useful forms, between TEX and the user, while TEX is running.* Thus if you look at a printed document and see that TEX put a certain box in the wrong place, that is useful information, but it doesn't match my definition of dialog because the communication didn't take place while TEX was running.

To give another example, a table macro package written by Ray Cowan, that I encountered under the name `tabls.sty` (an adaptation for LATEX), has the unique feature that you don't have to type a preamble line setting up the format of the columns in a table. The format is determined automatically by the contents of the table. The number of columns are then reported on screen while TEX is running. I classify this as dialog, even though TEX doesn't stop to check for any response, because I believe Cowan primarily envisioned the number-of-columns message as being read by the user while TEX is running, to see if the reported number of columns matches the intended number of columns. In the case of a minor discrepancy the user can just make a mental note to check the input file later for proper syntax; but in the case of a serious discrepancy (*93 columns??! Whoa!*), the user could press the interrupt key to break out of TEX and go fix up the table, before trying again.

On a more practical level, *dialog with TEX* usually involves sending and receiving messages using the `\message`, `\write`, and `\read` commands. 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 what Knuth means by "dialog" is the standard sort of programming tasks that involve providing information to the user, displaying menus, asking questions, and handling user responses. It's easy to identify a number of fairly obvious principles that should be followed when writing such dialog into a program:

[0] An extended version of this paper and some example macro files are available on request from the author.

1. When asking a yes/no question, the user should be able to enter `y`, `yes`, or even `ye`, in lowercase or uppercase, 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 as easy as possible to select.
3. Users' answers should be repeated back to them, so that they can verify that the answer taken in by the program is indeed the answer that was typed.
4. The user 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.
5. When practical, users' answers should be checked to make sure they're not nonsense; for example, if the program requests an integer, it should check the response to make sure the user didn't enter something else entirely. An example of this is the LaTeX option file `checknum.sty` that was published by Brian Hamilton Kelly in UKTEX, vol. 1, no. 1 (4 January 1991) in response to a query.
6. When giving information to the user, it should be provided in the *best possible form*, where the meaning of *best possible* should 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 register such as `\vsize` to the user will produce the value in points, down to five decimal places. For an average author it would usually make more sense to convert the value to inches or centimeters, whereas for a typographical designer or compositor it would usually make more sense to convert the value to picas, before it is reported to the user. Depending on the unit chosen, it should also be rounded to the nearest whole unit or tenth of a unit or something sensible that will avoid burdening the user with irrelevant precision.

It appears that Knuth's words "it's easy" weren't intended entirely literally, since the whole section where they appear is marked off with double dangerous bend signs; furthermore, the very next thing after the example mentioned above is Exercise 20.18 — marked with a double dangerous bend sign — 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.

The line-ending space is only one of a number of complications that can hamper the efforts of macro writers to write dialog into their macros. The aim of this article is to provide solutions for some of the complications.

Basic capabilities of TEX for sending and receiving messages

Tables 1 and 2 list the various means in TEX for sending messages to the user, and for the user to reply.

Table 1

Sending
`\message`
`\errmessage`
`\write`
`\show`
`\showthe`
`\showbox`
`\showlists`

Table 2

Receiving	prompt displayed by TEX
`\read`	`\controlseq=`
error message interaction	?
show message interaction	?
input file not found	`Please type another input file name:`
interrupt key	none

Notice that there are a few related features of TEX, e.g., `\errhelp`, that have their place in communicating something to the user, but that are dependent on the commands listed in the table: for instance, the user won't normally see `\errhelp` except by way of `\errmessage`. It is merely a temporary storage area for help messages, rather than a function that can be used to make something happen.

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. 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. If a message is more than one line long, and the macro writer does nothing to break it up into shorter pieces, TEX will break it up without regard to the contents of the message, even splitting words, using the maximum number of characters allowed per line (*max_print_line*, compiled into TEX) as the only line-breaking criterion. To obtain better line breaks, the macro writer can use the current newline character, determined by the value of `\newlinechar`, provided that it is a printable character. It is an idiosyncrasy of TEX that control characters, such as the `^^J` that is the default newline character in $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX and LATEX, cannot be used to produce new lines in the argument of a `\message` command, whereas in the argument of a `\write` command they work fine.

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. In other words, the command `\errmessage{Surprise}` produces

```
! Surprise.
```

on screen. Actually it produces more than that — it 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).[1] Newline characters in the argument of `\errmessage` operate the same as for `\message`.

`\errrmessage` is also noteworthy for the error recovery choices offered by TEX at the ? prompt. Among other things, choosing the 'h' option at the ? prompt will cause TEX to print on screen the current contents of the token register `\errhelp`.

The `\write` primitive The `\write` command, like `\message`, basically just prints a message on screen. But communication with the user is not the primary purpose for which `\write` was designed. Its primary purpose is saving index or table of contents information, with the associated page numbers, in a separate 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 and only executed when the page is actually shipped out, i.e., after the actual page break has been determined. To avoid such postponement, `\write` must be used with the `\immediate` prefix. But don't forget completely the link between `\write` and `\shipout`, because sometimes it's useful to leave off the `\immediate`. For instance, if you are working on page breaks in a long document and want to find out, without previewing or printing, if a non-forcing pagebreak command had the effect that you wanted, you could insert a non-immediate `\write16` just before and just after the intended page break:

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

(or, in LATEX, `\pagebreak[3]`)

```
\write16{After the attempted pagebreak.}
```

The message from a non-immediate `\write16` will appear just before the closing] of the [] pair that enclose the relevant page number. So if all went well, one message 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] ...
```

The `\show` and `\showthe` primitives The `\show` command, used for showing the current meaning of a control sequence (or indeed of any valid 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

?
```

As with `\errmessage`, TEX displays the surrounding context of a `\show` command; it also offers the same error recovery opportunities, except that you can't access `\errhelp` through the 'h' option, after a `\show` command.

The `\showthe` command is like `\show`, but is applied to certain kinds of things such as the names of 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 we defined earlier):

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

?
```

The `\showbox` and `\showlists` primitives The commands `\showbox` and `\showlists` are similar to `\show` in what they produce on screen. Because of their specialized nature they don't have much relevance to the main theme of this paper, but once

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

again don't forget about them entirely, because in certain narrow applications they might be just the ticket.

Error recovery

After an error message or a `\show` command, the user is presented 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:

```
   ...
? ?
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.
?
```

Because their main use is in recovering from errors, it is convenient to call these *error recovery options.* The insertion and token-skipping options, however, are potentially useful for other things besides error recovery.

An example of a typical use of these error interaction possibilities is given in Example 1.

Notable examples of dialog with TEX

To flesh out my definition of *dialog with TEX* let me give some examples from widely available macro files. This will also help to illustrate some of the typical difficulties.

Comment in `hyphen.tex`

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
```

(Here I have split the comment into two lines because of the narrow column width, but in the original, the comment is all on the same line.) Ordinarily the macro writer can't use comments to communicate with the user, because comments within the text of a macro disappear 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.
1.2 \patterns
              { % just type <return> ...
?
```

This idea could be useful in other applications. For example, the file `lfonts.new` of the Mittelbach/Schöpf font selection scheme (LATEX version) has a statement of the form `\input fontdef.tex`, where the file `fontdef.tex` is often missing (intentionally) and the user is supposed to substitute another file name such as `fontdef.ori` or `fontdef.max`. A comment on the same line as the `\input` statement, listing the alternate possibilities, could save users a certain amount of flipping pages in the documentation:

Example 1: Example of 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 \hru...
1.120 \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 \hru...
1.120 \hline

?
```

All right, the opening curly brace has just gone by, so we are indeed at the beginning of the definition of `\hline`. We need to insert the `\\` that was forgotten, and also replace the two tokens `\noalign` and `{` that have slipped by.

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

```
\input fontdef.tex % Other possibilities:
                   % fontdef.ori, fontdef.max
```

`Amstex.tex`: `\printoptions` Example 2 shows the implementation in $\mathcal{AMS}$-TEX of a `\printoptions` command. This allows the user to choose a syntax check or 'galleys' run, instead of a full typesetting run, the advantage being an increase in processing speed. But our primary interest at the moment lies in the techniques used to present three choices to the user and read the reply.

This example 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 both be treated the same. The method used is the second method given in the answer to *The TEXbook*'s Exercise 20.19. One more noteworthy feature of `\printoptions`: it runs a loop that doesn't quit until the user gives an acceptable answer.

Other examples

- `latex.tex`: `\typeout` and `\typein` are examples of the kind of basic dialog tools that can be built into a macro package.
- `docstrip.tex`: The `docstrip.tex` utility by Frank Mittelbach is used to strip out comment lines from a documented macro file. It provides 'progress reports' in the form of a message containing a single % or . character, for each line as it is processed. This produces rows of percent signs and periods in a random pattern across the screen, as the documentation stripping process chugs along, which helps to alleviate the monotony of processing a large file.
- `checknum.sty`: It is often useful to check replies from the user to make sure they're valid. Brian Hamilton Kelly's checknum.sty (posted to UKTEX vol. 91 no. 1 (4 January 1991)) illustrates a technique for reading an integer from the user and making sure they did indeed enter an integer and nothing but an integer.
- `animals.tex`, `basix.tex`: These two files by Andrew Marc Greene (the former published in *TUGboat* 10, no. 4 as part of *TEXreation — Playing games with TEX's mind*, the latter in *TUGboat* 11, no. 3 as *BaSiX: An interpreter written in TEX*) are examples of macro files whose whole purpose is carrying on dialog.
- `Testfont.tex`: This file, written by Knuth for his own use in testing new fonts produced by METAFONT, contains a `\help` command — something that would probably be a good idea for every macro package.

Use of `\message` and `\immediate\write`

Any expandable control sequences in the argument text of a `\message` or `\write` command will be expanded. Consider Table 3. This expansion is usually useful, but occasionally it can be a hindrance, as for example if you want to include a ~ in the text. And generally speaking, if you want to mention any control sequence in the argument text, you'll have to use `\string` before the control sequence (and frequently `\space` after it, as well). Table 3 also illustrates the utility of `\noexpand` for this purpose (something which was pointed out to me by Michael Spivak).

Prompting and reading input

Let's return to the $\mathcal{AMS}$-TEX `\printoptions` example now. Since `\W@` is defined to be `\immediate\write16`, and we know from the earlier discussion that 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 new line instead of immediately after the colon. This brings to mind the question: what can we do if we want the user's reply to appear on the same line?

The way to do this, in general, is by using `\write` to send all but the last line of a prompt message, and use `\message` to send the last line. (Brian Hamilton Kelly's `checknum.sty` uses this idea.)

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

Dealing with the Control-M/space character at the end of a `\read` macro In `\printoptions` a separate macro `\S@`, `\G@`, or `\P@` is 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 whole problem here is that `\read` includes the Control-M character at the end of the user's response in the macro being read. Under normal conditions Control-M is converted to a space, of course, but another possibility — if the user just presses Return without typing any response — is that the Control-M will produce a `\par` token (following the general rule that an empty line is equivalent to `\par`). The best approach (IMHO) is to prevent the Control-M character from getting into the read macro in the first place. This can be done in two

Example 2: \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}
```

ways: setting the catcode of ^^M to 9 ('ignore'), or setting \endlinechar to −1.

But that 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 is necessary in order for \answer to be properly defined when the group ends.

The tests in \printoptions can, with this modification, be simplified to:

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

and the macros \S@, \G@, \P@ are now totally 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 appropriate spot. We have the opposite problem if the user typed more than one letter: on the true branch the extra characters will most likely cause spurious typesetting activity. As it happens, we can kill two birds with one stone, as we'll see shortly when we discuss default responses.

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@}%
```

I prefer a slightly more economical version of the same technique:

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

This may be a bit confusing at first sight. 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. (Only the 's' is affected because the other tokens are control sequences or nonletters.)

Notice that the auxiliary macros \Ans@ are no longer needed. To simplify the structure of macros using this uppercasing process, it could be embodied in its own macro:

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

Default responses One last refinement in \printoptions would be to provide a default response if the user's response is empty (i.e., they just pressed the Return key). The method I like involves an auxiliary macro like the LATEX macro \@car:

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

But since most of us are probably not particularly well acquainted with the arcane terminology of Lisp, let's call this macro \firsttoken instead:

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

(Using @ as the ending delimiter is pretty safe if we make sure that it has catcode 11 at the time