TEX as a callable function

Jonathan Fine
203 Coldhams Lane
Cambridge, CB1 3HY
United Kingdom
jfine@activetex.org
http://www.activetex.org

Abstract

Traditionally, TEX is run as a batch program. However, TEX can also be run as a daemon, with a callable function interface. This talk describes the opportunities and problems that follow from this new way of using TEX.

The TEX daemon

TEX is a reliable and high-quality typesetting program that runs on a wide variety of platforms. It is tremendously quick at typesetting, once it gets going. However, it can take a significant fraction of a second for TEX to initialise its data structures (say by loading a precompiled format file). The TEX daemon is a way of avoiding this start-up time, say by removing it from the traditional edit-compile-preview loop. Doing this makes Instant Preview and other interactive applications of TEX possible.

A *daemon* or *service* is a program that is running continually in the background, waiting to be called upon when required. Many Internet services, such as FTP and HTTP, are provided by daemons. A daemon supplies data, or provides a service such a printing, upon request. Many daemons are essentially stateless. In other words, one FTP request does not affect the results of another. Cookies provide states to HTTP, which would otherwise be stateless. For some daemons, such as a database server, safely recording changes of state is one of the main purposes.

By running TEX as a daemon, we avoid startup costs. When typesetting a single paragraph, this can give a speed-up of 30 to 50 times. It is precisely this speed-up that makes it possible to use TEX as the typesetting engine of interactive applications. However, care must be taken. Certain commands, when fed to TEX, can cause it severe indigestion, or worse.

For example, in plain TEX the command `\end` will in most circumstances cause TEX to terminate. (For LATEX the command is `\stop`.) This is desirable, sooner or later, in a batch program, but not in a daemon. This is a trivial example, but it shows that inappropriate input can damage or even terminate the TEX daemon.

Some other examples are more subtle. First of all, TEX's capacity is finite, and so can be broken. Feed in too many control sequence names, say by `\csname`, and TEX will run out of hash space. When processing a single document, this is generally not a problem. But if we want the TEX daemon to be up and running all day, and processing fragments from hundreds of documents, then TEX's inability to reclaim unused control sequence names could be a problem. (See bugs 422 and 493 in TEX's error log, and the surprises section in [3].)

The most subtle example I am aware of concerns hyphenation. The command

```
{\hyphenation{su-per-cal-i...}}
```

adds a word to the exception dictionary (for the current language). However, this dictionary is global. Exceptions do not disappear at the end of a group. Therefore, as exceptional hyphenations are added through the day, so TEX's state slowly changes. As a result, a paragraph that was broken into lines one way in the morning might be broken in another in the afternoon. TEX rightly has a reputation for giving the essentially identical results on identical input. For the TEX daemon to preserve this admirable quality, it must forbid new hyphenation exceptions, or find some way of reversing additions to the exception dictionary, or find some other way out.

However, these are minor problems. As a daemon TEX, has many admirable qualities. It does not crash, core-dump, or otherwise come to an abrupt end. It behaves in an entirely predictable way (even if sometimes it surprises us). It does not leak memory. Correctly fed, it will typeset pages all day, and still be as fresh at dusk as it was at dawn.

Although designed as a batch program, it is a remarkable testament to the soundness of its design and implementation, that it can also be run as a daemon.

TeX as a callable function

TeX takes as input a text stream, and its output is a stream of pages, encoded as `dvi`. (There are of course other inputs, such as fonts, hyphenation data, and macros.) The output `dvi` file consists of a preamble, then some pages, and then the postamble. Every font that occurs in the `dvi` file is defined twice, once immediately before its first use, and once in the postamble.

Almost all `dvi`-reading programs go immediately to the post-amble, in order to pick up this list of fonts. (Tom Rokicki's previewer for the NeXT platform is an exception. It can start previewing from page one.) From the same place, they pick up a pointer to the last page, and from there a pointer to the previous page, and so on. Once they have done this, they are in a position to perform random access on the pages in the `dvi` file. If the postamble were not available, to find and process say page 20, the program would have to process all pages 1 through to 19, firstly just to find page 20, and secondly to be sure to pick up all previous font definitions.

Clearly, if TeX is run as a daemon, it is not possible to wait for the postamble, before responding to a user request. Instead, we have TeX write to a named pipe or something similar, and place at the other end of the pipe a utility program, that parses the output `dvi` stream, and sends pages to the appropriate destination.

The author's program `dvichop` does just this. It cuts the output stream into small `dvi` files, and optionally sends a signal to the originating process. This design decision allows existing `dvi`-ware to work with the TeX daemon, at the quite bearable cost of each byte of the `dvi` being processed twice.

Another benefit of this approach is that it allows the behaviour of TeX as a callable function to be specified. In the previous section, we learnt that the TeX daemon should have a state, and that each use of the TeX daemon should leave it in essentially the same state. This imposes conditions of what we can feed to the daemon. The fixed state of the TeX daemon can be set up by a preloaded format file.

We can now define valid input to the TeX daemon to be input that possibly produces output `dvi`, but which does not otherwise change the state of TeX. The grouping commands already provided by TeX will go a long way towards enabling such input. We can also define the output of the TeX daemon, for any valid input. For simplicity, we assume that the file is `\input` by TeX, with certain fixed macros running before and after the `\input` command.

The output of TeX as a callable function is then the `dvi` file produced, from the initial state defined by the format file, after the given valid input is processed, and an `\end`, `\bye` or `\stop` command has been issued (or alternatively, have TeX make an emergency stop).

We have just defined, without reference to the implementation, the behaviour of TeX as a callable function. The above definition gives one implementation. Using the TeX daemon (say with `dvichop`) gives another, which will be much quicker on page-sized files.

What should now be clear is that the preloaded format file, and the creation of valid input, are crucial to any particular instance of the TeX daemon. Roughly speaking, there are four approaches to the problem of valid input. The first is the original definition, namely that it does not change the state of TeX. However, this definition can detect invalid input only after the event, which is too late. In addition, it does not give an interface to which a user of the TeX daemon can write code for.

The second approach is to ensure that any input whatsoever can be processed without changing the state of TeX. Ordinarily, TeX input has direct access to all primitive commands, and all macros, of TeX. This approach involves inserted a layer of TeX macros between the input file and the execution of macros. This is similar to the protection offered by a modern operating system kernel, which does not allow user programs to access the hardware directly.

Active TeX [1], which makes all characters active, is one way to do this. By definition, the only commands that user input can directly access are those tied to the characters. These commands in turn will produce and execute control sequences. However, this production of control sequences is under the control not of the user, but of the system's layer of active characters.

The third approach is to use specially designed software to filter out bad input, and translate into standard TeX calls. Using an XSLT script to generate LaTeX is an example of this approach. The input is not creating TeX calls directly, and the LaTeX that is created can be tightly controlled. In many such cases, in principle if not in fact, a formal specification could be given for the possible outputs. Such would be a contract between the creator of input to TeX and the provider of TeX macros.

Here is an aside. In the author's view, even though it has many capabilities and wide range of libraries, LaTeX is not a suitable language for this purpose. This is because it has many exceptions, and because it was not designed with purposes such as machine generated input and output supporting interaction in mind. For example, many of its facilities rely on category code changes. In addition, in some cases '`[`' has a special meaning. The author, who thinks himself an expert, from time to time falls into such gotchas.

Using XSLT to produce LaTeX is today a sensible short-cut for producing print from XML. However, surely even its most devoted supporters will agree that for LaTeX to be an internal interface format for a web browser or a WYSIWYG word processor would be a triumph of inertia over sound design.

The fourth approach is to ignore the problem, and provide a means for restarting the TeX daemon if it gets into trouble. Instant Preview uses this approach, largely by running TeX in `\errorstopmode`, and using a judicious `\outer` macro to stop errors propagating from outside user area. Of course, if the user input contains

```
\global\let\let\undefined
```

then the user will get what he or she deserves (which is no sensible output).

Finally, there are a number of technical problem related to the implementation of TeX as a callable function. In this paper, we will simply note three of them. The function call will send a string to TeX, and expect to get `dvi` back in return. The first problem is not to return until the `dvi` is available, or in other words to wait until TeX is done. The second is not to block, or in other words have both TeX and the function call both waiting for the other to supply input. The third is to handle contention, or in other words, multiple requests overlapping in time. These are standard problems in client-server programming, and even if not by the TeX community, solutions are well-known.

Random Access Typesetting

Several developers, besides myself, have been writing software that gives more or less immediate visual feedback to the person editing the document. We all face a common pair of problems: How to slice out of the document a region to be typeset, and how to initialise TeX so that it can correctly process this slice. These are leading problems in random access typesetting, a term which we will now define.

Most users of TeX know what random access previewing is. It means loading a `dvi` file quickly, and being access any page in the previewed document quickly. Recall from the previous section that a `dvi` file has special structure, designed specifically to support such operations. Random access typesetting is being able to go to any point in the source document, and to quickly typeset some region surrounding that point. Page breaks (and page numbers) present special problems. However, most of the time getting these just right is not so important to the user, so we will ignore this problem.

An extended form of random access typesetting is where not only the point in the document is random, but the choice of the document itself. This is the sort of task a web browser has to deal with. Later in this section we will see that it presents a new range of problems.

A valid LaTeX document has structure, understood by the LaTeX macros. Provided the document is not too unusual, it is possible for another program, such as a collection of Emacs macros, to divide the document into blocks that can be typeset individually. However, such software is fragile. Careless keyboarding, or even well-designed user defined macros, can break such a system. The Perl scripts that convert LaTeX to HTML face similar problems, with which their users are familiar.

This brings us to the second problem, which is properly establishing the context in which the document fragment can be typeset. Knowing the section numbers and theorem numbers right is one part of this problem. Knowing the formatting context (abstract, footnote, body text etc.,) is the other.

The author's preferred solution to this problem is to make the document being edited responsible for solving this problem. In other words, one writes a script, based say on LaTeX to HTML conversion, that adds to the document what we can call belays. (In climbing, a belay is a point of safety, to which a climber attaches a rope.) Each belay should include section, theorem and other such numbers, as well as a statement as to the typesetting context.

The belay data need not be stored in the document itself. It could be stored in the `aux` file (and be made available to LaTeX via a `\csname` lookup). In this way, all that needs to be added to the document are commands such as `\belay{27}`. This can be done, say, with a Perl script. In addition,

```
\usepackage{belay}
```

in the preamble will cause an initial typesetting run to write belay information to the `aux` file. At present, `belay.sty` is vapour-ware. Thanks are due to Simon Dales, who suggested the term 'belay', and to Johan Andersson, for sharing with me a prototype he set up along similar lines.

At present, LaTeX is set up for sequential batch typesetting, rather than random access typesetting. This causes various problems. For example, in the article style file, the `\maketitle` command redefines itself to `\relax`. There may be other similar gotchas.

Here is another class of problems. Each LaTeX document has a preamble, and even when two articles are for the same journal, they very often have different preambles. Even if two preambles differ only by the use of a package, this presents a problem, for at present LaTeX allows packages to be loaded only in the preamble, and not after typesetting is underway. When two documents number theorems in different ways, or have different user defined macros, additional problems are created.

In light of this, the author believes that it is not practical for two random LaTeX documents to share the same TeX daemon. The author also believes that with some judicious changes, LaTeX can successful be used for random access typesetting of a single document. Some of these changes relate to error recovery. For example `\scrollmode` and then `\section` without any arguments produces an isolated ']' character in the output `dvi` file.

Macros for use with the TeX daemon

Without macros TeX is unusable, because its primitive commands are, well, so primitive. A macro package such as plain or LaTeX does several things. Some of these are: (1) It loads fonts and hyphenation patterns. (2) It defines a custom input syntax. (3) It sets typographic parameters such as the measure, and provides commands for changing these values. (4) It sets up commands for the typesetting of mathematics. (5) ditto, but for tables. (6) It defines an output routine. (7) It takes care of page, section, equation and other numbering. (8) It writes out index and table of contents information.

Both plain and LaTeX were written for batch use of TeX. The user creates a document, which is submitted to the TeX compiler. TeX then returns, hopefully, a `dvi` file and a log file. The user then studies both. The log file records both parse errors (such as misspelt control sequences) and typesetting difficulties (such as overfull boxes). Except for the media and the turnaround time, the situation is the same as submitting punch cards to a mainframe.

The TeX daemon is a completely different setting. TeX the program is already running, and we would be well pleased if the user's input left the daemon as she would wish to find it. The robust solution is to filter out user errors, particularly those that harm the TeX daemon. As stated early, this can be done in TeX macros, but other ways may be better.

At a Question and Answer session (1996, Amsterdam), Piet van Oostrum asked Don Knuth about TeX's macro programming language [5, p648–9].

The reader is encourage to read the whole of his response, and indeed all the Q+A sessions. Here we summarise points of special interest. (1) Don wanted to avoid introducing "yet another almost-the-same programming language" for TeX. (2) Many features were added "only after kicking and screaming" from users. (3) Users wanted to "put low-level things in at a higher level." (4) Don expected that "special applications would be done by changing things in the compiled code." (5) Don wanted to write just a typesetting language, and not a programming language as well. (6) He also said "if there were a universal simple interpretive language that was common to other systems, naturally I would have latched onto that right away."

I have spent several hundreds of hours writing clever TeX macros for doing low-level things (like parsing SGML), and collectively the authors of big macro packages have probably spent even longer doing this sort of thing. I have learnt to accommodate myself to the limitations of the language, and how to make the best use of the features it does provide. Some of these programming tricks are ingenious, and even elegant. However, I think it is time for a change.

One of the main reasons for this is that clever TeX macro code can only be used with TeX (or its successors), and conversely clever or even just solid and reliable code in other languages cannot be used inside TeX (although there may be some workarounds). Another reason is that other languages can be more efficient, for both the programmer and the computer.

Here is an example. To parse a text string is to analyse its structure, to break it down into tokens arranged in some way. A natural language parser will find the subject, object and verb in a sentence. Parsing is a non-trivial activity, upon which further processing depends. LaTeX contains a parser. But because it is written in TeX macros, it cannot be shared with other applications.

There is an alternative to TeX macros, not mentioned by Don in his answer. This may be because he takes it for granted. Literate programming is an example. Here a custom program (`WEAVE`) takes a document that TeX does not understand, and produces from it an input file for TeX. Some XSLT scripts are another example.

Since TeX was written in the 70s and 80s, there have been new interpretive languages, some of which are widely used. There's Perl, Python and Ruby.

There's Java and JavaScript. There's Scheme and Guile. And there's Visual Basic and C♯.

Are any of these, in Don's words, "a universal simple interpretive language [...] common to other systems"? But is this a judgement of Paris? How can we use one, without offending the rest? Without starting a language war? This is a difficult problem, and like all human problems, its solution requires both good-will, good ideas and a measure of wisdom.

The author suggests that TEX macros be used where only TEX macros will do, or when required for efficiency. External C/C++ modules for external modules, such as an XML parser. Scripting languages for control of style and placement, and for application specific code. A typical application today might be to use Perl to retrieve records from a database, and send them to TEX for typesetting. Under the new scheme, the application would the same, except the TEX stream would be written using a Perl interface module, rather than directly.

We do have at least one good example to follow. The Tk graphics toolkit, was developed by John Oosterhout as a companion to the Tcl scripting language. Since then interface modules for Perl and Python have been written, that allow these languages to make Tk calls. Even though Perl, Python and Tcl have different syntax, they all interface to Tk in much the same way. Perhaps something similar could be done with TEX.

dvi-ware for use with the TEXdaemon

Interactive applications place new demands on `dvi` files, and on the programs used to process them. The range of interactions with the previewed page is vast, and at present hyperlinks and in some cases marking of text is all that is supported.

Indeed, most programs for displaying `dvi` on the screen are both by name and function previewers. A preview is a depiction of something that is not yet present. In our case, it depicts a typeset page that we might or might not choose to print. However, in many interactive applications what is being displayed is not a preview, but the object itself. Indeed, in journalism it is common to print a screen-shot of a web site, so as to preview (in print) the object of interest, namely the web site.

`dvichop` is not a complicated or particularly specialised `dvi`-processor. Before writing it, I looked at the source for a good number of the free `dvi`-ware programs, hoping to find some code I could use. Sadly, I found nothing that was helpful. So I had to write the program from scratch.

By and large, there are three types of `dvi`-ware: utilities, printer drivers and previewers. So far as the author knows, Anselm Lignau's TkDVI is the only `dvi`-processing program that can be used as part of a scriptable interactive program.

Applications

Here we describe various applications already or being developed. In theory, and hopefully in the long term, goals such as a web browser that supports mathematics, and a WYSIWYG editor for TEX are possible. Here, the focus is on small, simple and largely self-contained projects that are immediately useful, and which take us forward.

TEX showcase One of TEX's great strengths is its line-breaking algorithm. Because it optimises globally, change at the end of the paragraph can move the first line break. We all know this in theory, but seldom observe this in practice. Because the TEX daemon provides instant feedback, it is now possible to write an application that showcases TEX's line-breaking algorithm. In other words, as one changes parameters and perhaps content, so one sees the typeset paragraph change.

Other parts of TEX, such as the mathematics and table typesetting, can be showcased in the same way.

Interactive courseware Newcomers to TEX often require a lot of visual feedback, to reinforce in their minds the connection between the characters they type and the words and formulae that appear on the page. Using the TEX daemon as the typesetting engine, interactive courseware can be built, that helps beginner to learn LATEX, or whatever their favourite macro package is.

Instant Preview This was demonstrated at EuroTEX 2001 [2]. It works as follows. Suppose the active buffer is in Preview mode. Then at every keystroke a small region, that contains the part of the buffer that is visible, is sent to the TEX daemon, and thence displayed in an `xdvi` window. This provides Instant Preview.

References

[1] Jonathan Fine, Active TEX and the DOT input syntax, *TUGboat*, **20**, (1999), 248–254

[2] ———, Instant Preview and the TEX daemon, *EuroTEX 2001 Conference Proceedings*, 49–58

[3] Donald E. Knuth, The Errors of TEX, *Software — Practice and Experience*, **19** (1981), 607–685. (Reprinted in [4])

[4] ———, *Literate Programming*, CSLI (1992).

[5] ———, *Digital Typography*, CSLI (1999).