♢ ♢ ♢

TEXlib: a TEX reimplementation in library form

Giuseppe Bilotta

Introduction

I first came across the need for a TEX in library form when I was thinking about developing a graphical real-time front-end to TEX (the TEXPerfect project, more info at `http://texperfect.sourceforge.net`). A quick survey (on the `comp.text.tex` newsgroup) showed that other projects could have benefited from a library providing TEX typesetting capabilities, and I thus decided to develop TEXlib as a separate project from TEXPerfect. A "call for developers" on the same newsgroup provided the project with developers/consultants/helpers.

An analysis of the current opinion on TEX and its future added another aim to the TEXlib project: since we're reimplementing TEX, why shouldn't we take the occasion to break through TEX's limitations?

The current status of TEX's "future" is the following. We have some extensions:

- ε-TEX, mainly concerned with TEX's parser (eyes and mouth), but also providing right-to-left typesetting;
- Omega, mainly concerned with TEX's typesetting routines (guts), but also with an innovative input parser (OCPlists);
- pdfTEX, mainly concerned with TEX's output routines, but also prodiving new typesetting features (one for all: hanging characters);

and some new implementations:

- *NTS*, in Java, close to its first version;
- ANT, in Scheme, still in a very early stage (but still more complete than TEXlib`;-)`.

More than once it has been suggested that the three TEX extensions should blend into one; well, TEXlib might as well be the point of convergence. The reasons behind such an expectation are mainly

- the library form of TEXlib: while a library can be easily used with a command line interface, it is much harder to let a command line driven program act as a library;
- being the youngest and less-formed TEX-based project, TEXlib can deal with all the issues of integration of extensions, without requiring changes to (not-yet-existant) sources. This assumes that the ε-TEX, pdfTEX and Omega developers are willing to provide feedback and suggestions on how to integrate the various features provided by the different extensions.

Library structure

The library will be split in three parts: input parser modules (eyes & mouth), typesetting module (guts), output modules. The three parts will be kept as separate as possible (thus allowing things like an XML input parser with an Omega typesetting engine and producing PDF output).

The typesetting module interface will be public, so that custom input parser and output producers could take advante of TEX's typesetting capabilities. The typesetting module will basically provide two functions: the paragraph builder (accepting a horizontal list and returning a vertical list) and the page builder (accepting a vertical list and returning two vertical lists).

Vertical and horizontal lists will be built by the input parsing modules, and can be sent to the output modules to produce "real" output (DVI pages, PDF documents, some other format specifically tuned for real-time preview, etc). The library will provide default input parsers and output producers.

Assuming the library-provided modules will be used, the following is more or less how a typical session of TEXlib would run.

1. TEXlib is loaded (if not loaded already).
2. A TEXlib "context"[1] is initialized, providing a format file, and various settings, such as: which extensions are allowed, what kind of input is provided (TEX, XML), what kind of output is expected (PDF, DVI, memory output), how many pages to "cache" in memory, etc.
3. The main function will be provided with the address of the buffer containing the data.
4. Variables for the context are initialized.
5. Typesetting Loop.
6. Feedback Loop.

Library loading and library instantiation are kept separate, not only to allow the library to be shared among clients, but also to allow the same client–library link to make use of different TEXlib contexts (useful if typesetting a mixed TEX/XML document, for example).

TEX input parsing approach

While the current TEX extensions deal with *what* TEX does, the main concern of TEXlib is *how* TEX is supposed to do it.

Currently, TEX works in a sequential way: source code is input one line at a time, and the data is sequentially processed to ship out some kind of output (DVI file, one page at a time, plus various auxiliary files, depending on the format used).

Much of this processing mode is somehow required by TEX's *embedded macro* (programming) capabilities (or, conversely, TEX's macro capabilities were built with this workflow in mind). This means that no *revert* is possible: once a change (`\def`, catcode change, etc) has been made, the only way to "roll back" is through another

[1] "context" is the internal name of a library instantiation.

"forward" change, restoring the previous value (which must have been saved somewhere, usually in another macro).

Since the main application of TEXlib is TEX real-time editing, and since editing (and especially reviewing) happens in a non-sequential way, we need a way to allow moving backward and forward through the source.

Mainly, we can consider two approaches:

1. ONE-STACK-PER-THING-TO-BE-SAVED APPROACH.
   The idea behind this approach is to push the old value each time a change is made, and then pop it when rolling back.
2. CHECK-POINT STACK APPROACH.
   This approach can be thought of as "intermediate dumping": fix a check-point (say, at page ship out), and push/pop *all* the values (also the unchanged ones) each time the user crosses the check-point.

Functionality. Pros and cons

Let's consider approach 1 first. The technique works as follows. A token is examined and 'executed'. If the execution (complete macro expansion or execution of a primitive) changes some values, the library stores, at the end of the execution, the original values together with the new ones. This allows easy back-tracking, and restart of compilation from an arbitrary point.

The largest problems are memory usage (but a comparison between this method and the next one would need some real-world cases) and the complexity connected with `\let` and `\def` when applied to or otherwise influencing the same token that causes the change. Currently, the best idea I can think of is to simply wait until macro expansion and argument scanning ends, before saving the values, but I still couldn't think of a robust way to implement such an idea.

Let's now have a look at approach 2. Probably the best place to insert a checkpoint is at page shipout. The library then acts this way: it parses data until it fills the page cache; then it waits for client feedback; if data needs to be re-parsed, the library re-enters the typesetting loop.

1. Typesetting Loop.
   a. (Parsing and typesetting) If no data available goto 1.e else read next token and execute it.
   b. If shipout goto 1.c else goto 1.a.
   c. (Shipout) Save $checkpoint = (line, col), page, memory\_dump$. Increase $cached\_pages$.
   d. If $cached\_pages = max\_cache$ goto 1.e else goto 1.a.
   e. Tell client that we finished our job.
2. Feedback Loop.
   a. The client tells the library where the cursor is.
   b. If cursor crosses a *checkpoint*, reload $memory\_dump$ for the entered page.
   c. If a change has been made (a token has been inserted), goto 2.d else goto 2.a
   d. Set $line, col$ to that of the latest *checkpoint* and goto 1.

This approach is relatively easy to implement, but quite memory consuming (consider e.g. that a typical ConTEXt format file is between 4 and 5 megabytes in size, and this

amount of memory should be allocated for each cached page). Also, it is quite slow when rolling back before the first cached page, since in this case typesetting would have to start right from the beginning of the file (when the first cached page is around page 100, this would mean that we need to retypeset $100 - max\_cache$ pages). These two problems could be minimized with

1. dynamic memory allocation;
2. "unbalanced" cached pages.

The idea behind 1 is that memory should only be allocated when needed, thus giving smaller memory hits (and higher performance) for typical jobs, while still allowing heavy jobs to be done without reinitializing the library.

The idea behind feature 2 is to keep more "back" pages than "forward" pages in cache. For example, if there are 10 cached pages, the current page is likely to be the 8th or 9th cached page (provided that we are past page 8 in the document). If the user is scrolling backwards, the library will restart compiling before the user hits the 1st cached page (say, when the user gets to the 5th cached page), discarding "forward" pages (say, from the 7th to the 10th) and it will stop compilation three pages before the first cached page.

The various settings (number of cached pages (10), number of back (7) and forward (2) pages and the discarding treshold (5)) should be user configurable, possibly at runtime. A future version might have auto-detection of "best" suggested settings.

Another shortcoming in this approach, at least when fixing checkpoints at page shipout, is TEX's asynchronous page shipout. When a page is actually shipped out, TEX can already be quite a few source lines past the last source line on the page being shipped out. A possible solution could be to save two source coordinates instead of one: the $(line, col)$ pair of the data that caused the shipout, and the $(line, col)$ pair of the last data contained in the shipout.

Of course it is to be seen if there is some way to determine the last data shipped out, and the parent $(line, col)$ coordinate.

This problem is tightly connected with the synchronization of source and view. Consider that it was TEXPerfect that pushed me into developing TEXlib. Since TEXPerfect will likely run in split-view, with the output in the upper half and the source in the lower half, we need a way to synchronize cursor positions in the view with cursor positions in the source, and the synchronization has to be as precise as possible.

Synchronization

Synchronization needs a continuous feedback between client and library. On one side we have the client, which provides the source, the current position within the source, the modification status. On the other side, the library provides the output (in the specified form) and its status. But there is another important kind of feedback that the library can provide to the client (and we will see shortly why it is important): tokenization of the input lines. This means that for each input line read, the library should return where each token starts, where it ends and which subsequent tokens are being "eaten up".

Why is this important? Let's consider the first level of synchronization: source specials. At least for the current page (but possibly for each cached page) the library should know the originating $(line, col)$ coordinate for each output bit (character, rule, glue).

This can be memory-optimized by taking advantage of one-to-one correspondance: for example, in the case of a paragraph containing only characters, it is only important to know where the first letter originated from.

But there are cases of multiple tokens providing one or no bits of input (think of multiple spaces, or some kind of assignments), and conversely of single tokens providing more than one bit of input. To make things more complicated, most command tokens are multi-letter, and they can take arguments.

The idea is then to inform the client about this. Thus, tokens scanned during macro expansion will be given a particular status, so that the client knows which tokens will go "directly" to the output, and which should be considered as arguments of macros. The client can then take appropriate actions: for example, commands (both the command token and its arguments) could appear as a button in the source code window, and be skipped with a single keystroke while browsing (instead of requiring a keystroke for each character composing the token and its arguments).

Problems

There are some intrinsic problems that are inherent to a librarization of TEX, and they can be summarized as follows:

A. Error management

This answers the question: how to handle input-parsing errors?

Documents fed to the library could be split in two categories: 'hand-written' documents and 'machine-written' documents.

A hand-written document is simply a document written with a standard editor. A common source of error in such a case could be of the kind "`\hobx` instead of `\hbox`" (that is, all the kinds of error that arise from typos during source-writing).

Machine-written documents, on the other hand, are documents where commands are inserted in the source by the editor *only*, just like it happens, for example, with word-processors: the user selects *Italic* in the font properies (or presses an appropriate shortcut) and the client inserts the appropriate codes into the source.

Such a document will be free of typo-like errors (unless the editor has been badly programmed). But still, other kinds of error are possible (for example, fragile commands in moving arguments).

Since we are implementing a library, when such an error occurs the library would inform the client of the fact; acording to the spirit of batch processing in TEX, a suggested solution will be proposed. It is then up to the client to

choose what to do: consult the user, provide its own solution or simply enact what has been proposed by the library.

A useful option that the client *should* provide is "verbatim reparsing" from where the error occurred. For example, if the error was an caused by an undefined csname, the action would be to consider the csname to be a sequence of character tokens.

B. Input/Output management.
There are also other input/output issues. Two different approaches should be taken, distinguishing user I/O from file I/O.

Management of user input/output will be entirely left on the client side: the library will inform the client when user input is requested, and simply defer logging information to the client. The client is then free to report the logging information and the query to the users, or simply hide them. For example, in the case of a query to the user the client might ask the user for an answer, and then provide that answer as default each time the same query is met again during re-typesetting.

There are input/output issues connected with external file management, too. During re-typesetting of source files `\input`s and `\write`s referring to the same data will be met again and again. How is this to be managed?

First of all it is important to differentiate between user-provided `\input` (for example, when dealing with master documents and subdocuments) and "system" `\input` (for example, input of auxiliary files, as requested by recent formats like LaTeX and ConTeXt).

Since the client is the only one who knows if an `\input` is user- or system-provided, it should be left to the client to decide which `\input`s and `\write`s should be honoured and which not. The library will have to manage the input/output in a rational way, with complete knowledge on which data was output by which command, so as to be able to remove that data before insertion of the new data.

(This issue is still blurry and to be discussed. See also issue D.)

A separate problem is finally provided by the `\write18` primitive. But this will probably come up later (for example, if `\write18` is used to call Metapost, a possible solution is to just pass the request to the Metapost library, when it will be implemented).

C. Extensibily.
Should TeXlib be extensible? To what extent, how easily? This is a serious problem: I don't want to stimulate proliferation of many incompatible TeXlib extensions, while I still believe that time will show its limitations and thus the need to overcome those. This will mean that there will be an "official" TeXlib, with "official" extensions. (In other words, more ConTeXt-like development than a LaTeX-like development).

D. "Backward" compatibility and auxiliary files.
TEX format files use auxiliary files to pass information between a compilation and the next. Such auxiliary files are created during typesetting, sometimes post-processed, and finally re-input on subsequent compilations. Most of the time the data stored by auxiliary files is data provided later in the document but physically used earlier.

Such a way to pass information back and forth is required in sequential data parsing, and cannot be easily overcome when subverting the sequential paradigm while still keeping source compatibility.

There are though some things that can be done to solve some issues.

One issue is, for example, the (possible) need to change the stored data each time the writing command is issued. This might lead to physical abuse of storage supports (disks), and can be circumvented with intelligent data analysis (storing the data if and only if) and file caching (keeping the auxiliary file in memory instead of on disk —this requires the library to know which files are auxiliary ones).

Another issue is the actual usage of the data stored in the auxiliary file. This data is usually input at the beginning of the document, and this gives some problems.

First of all, it is useless to re-input the same data each time the user crosses an input command: a more intelligent way to deal with this is to check if the data has changed and act consequently.

The second important issue is the following: assume that the auxiliary file is actually input when needed (for example, when the cursor enters the table of contents), the same input may create differences in subsequent pages, thus producing differences in the auxiliary file, and possibly (in case of non-converging changes) cause a library lock-up.

I propose the following solution (inspired by the way a famous word-processor works). During normal editing/previewing auxiliary files are not dealt with, and input/output requests to it are simply ignored. At the user's request, though, a series of *sequential* compilations take place (the document is *generated*). The information collected during these sequential compilations is then stored in an auxiliary file and used.

Future ideas

A. Convergence of TEX extensions.
As mentioned in the introduction, the "planning" status of TEXlib encourages it as a point of convergence for TEX extensions. Cooperation with the developers of the other TEX extensions will render this actual.

B. Librarization of TEX's friends (*at least* Metapost).
The TEXlib project is not involved with TEX alone, but with the whole family. Librarization of the other members of the family —or of their successors— will

allow a previously unseen integration of the components —with all the power that comes from it.

C. Possible integration of TEX and Metapost in library form.
Library Metapost is a top priority (after TEX itself) of TEXlib; Among other reasons, because of the impressive power derived from the intregration of TEX and Metapost (it is even possible to emulate a poor-man's Omega, at least when referring to multidirectional typesetting capabilities).

D. Further extensions of TEX.
TEX is finally showing its age. While still outperforming other similar or related programs (from word processors to desktop publishing programs) in many aspects, there are features which simply cannot be implemented robustly without providing new core features. Some (maybe most) of these features can be implemented through the use of specials and appropriate output postprocessors, but native implementation of them could make the whole thing robust, standard and fast.

Some proposed extensions are the following:

- ⋆ *Native color support (and other attibutes).*
  This has been discussed (and will be discussed again when the time comes) on the TEXlib developers mailing list (and no conclusion has been reached). We came to the conclusion that it could be nice to "load" each node type with attributes not directly related to typesetting. Color is an example of such an attribute, since a character is put in the same place whatever its color is.
- ⋆ *Multiple reference points.*
  This has been requested by Hans Hagen, to ease and make more robust the management of `\vtop` and similar boxes.
- ⋆ *Code tables management.*
  Most of the internal code tables (catcodes, lccodes, uccodes, sfcodes etc.) of TEX (and of some of its extensions) have to be manually changed value-by-value each time such a change is needed. Internal support for saving/restoring (partial) code tables would speed up things like font- and language-switching. This feature probably needs to be made cooperative with Omega's ocplists.