The design of the HINT file format

Martin Ruckert

Abstract

The HINT file format is intended as a replacement of the DVI or PDF file format for on-screen reading of TeX output. Its design should therefore meet the following requirements: reflow of text to fill a window of variable size, convenient navigating of text with links in addition to paging forward and backward, efficient rendering on mobile devices, simple generation from existing TeX input files, and an exact match of traditional TeX output if the window size matches TeX's paper size.

This paper describes the key elements of the design and motivates the design decisions.

Why do we need a new file format?

The first true output file format for TeX was the DVI format [3]. When PostScript became available, it was soon supplemented by `dvips` [7], and now, most people I know use `pdftex` to produce TeX output in PDF format. There are two good reasons for that: partly, the PDF format is a near-perfect match [4] for the demands of the TeX typesetting engine, but first and foremost, the PDF format is in widespread use. It enables us to send documents produced with TeX to practically anybody around the globe and be sure that the receiver will be able to open the document and that it will print exactly as intended by its author (unless a font is neither embedded in the file nor available on the target device) .

But the main limitation of the PDF format is its inherent inability to adapt to the given window size. For reading documents on mobile devices, the HTML format is a much more convenient format. Part of the concept of HTML is a separation of content and presentation: the author prepares the content, the browser decides on the presentation — at least in principle. It turns out that designers of web pages spare no effort to control the presentation, but often the results are poor. Different browsers have different ideas about presentation, users' preferences and operating systems interfere with font selection, and all that might conflict with the presentation the author had in mind.

When it comes to ebooks, the popular epub format [2] is derived from HTML and inherits its advantages as well as its shortcomings. As a consequence, ebooks when compared with printed books are often of inferior quality.

What is needed is a document format which meets the demands of the TeX typesetting engine and that gives the author as much control over the presentation as possible but still can adapt to a given paper format — be it real or electronic paper. Building on previous work [8, 9], these two design objectives guided the development of the HINT file format.

While the TeX typesetting engine, its internal representation of data, its algorithms, and its debugging output, was the driving force of the development of the HINT file format, giving the whole project its name (the recursive acronym for "HINT Is Not TeX"), the result is not limited to the TeX universe. In the contrary, it makes the best parts of TeX available to any system that uses the HINT file format.

Faithful recording of TeX output

At the beginning of the design, the primary necessity was the ability to faithfully capture the output of the TeX typesetting engine.

To build pages, TeX adds nodes to the so-called "contribution list". The content of a HINT file is basically a list of all these nodes, from which a viewer can reconstruct the contributions and build pages using TeX's original algorithms. So with few exceptions, TeX nodes are matched one-to-one by HINT nodes.

Of course, we need characters, ligatures, kerns, rules, hlists and vlists; and as in TeX, dimensions are expressed as scaled points. But even a simple and common construction like `\hbox to \hsize {...}` requires new types of nodes: this is a horizontal list that may contain glue nodes and has a width that depends on `\hsize` which is not known when the HINT file is generated. To express dimensions that depend on `\hsize` and `\vsize`, HINT uses linear functions $w + h \cdot \texttt{\hsize} + v \cdot \texttt{\vsize}$, called *extended dimensions*. Linear functions are a good compromise between expressiveness and simplicity. The computations that most TeX programs perform with `\hsize` and `\vsize` are linear and in the viewer, where `\hsize` and `\vsize` are finally known, extended dimensions are easily converted to ordinary dimensions. Necessarily, HINT adopts TeX's concepts of stretchability, shrinkability, glue, and leaders.

One of the highlights of TeX is its line breaking algorithm. And because line breaking depends on `\hsize`, it must be performed in the viewer. But wait — an expensive part of line breaking is hyphenation and this can be done without knowledge of `\hsize`. So HINT defines a paragraph node, its width is an extended dimension, and all the words in it contain all possible hyphenation points in the form of TeX's discretionary hyphens. To maintain complete compatibility between TeX and HINT, two types of hyphenation points had to be introduced: explicit and automatic. TeX uses a three pass approach for breaking lines: In the first pass, TeX does not attempt automatic hyphenation and uses only discretionary hyphens provided by the author. Likewise HINT will use in its first pass only the explicit hyphenation points. Given the same value of `\hsize`, TeX and HINT will produce exactly the same line breaks. In a paragraph node, HINT also allows vadjust nodes and a new node type for displayed formulas to make sure that the positioning of displayed equations and their equation numbers is exactly as in TeX.

The present HINT format also has an experimental image node that can stretch and shrink like a glue node. Therefore, images stretch or shrink together with the surrounding glue to fill the enclosing box. The insertion of images in TeX documents is common practice. But TeX treats images as "extensions" that are not standardized. In a final version of HINT, I expect to have a more general media node. I think it is better to have a clearly defined, limited set of media types that is supported in all implementations than a wide variation of types with only partial support.

One node type of TeX that is not present in HINT is the mark node. TeX's mark nodes contain token lists, the "machine code" for the TeX interpreter, and for reasons explained next, HINT does not implement token lists.

Efficient and reliable rendering

On mobile devices, rendering must be efficient and files must be self-contained. To meet these goals, the proper foundation is laid in the design of the file format.

The most important decision was to ban the TeX interpreter from the rendering application. A HINT file is pure data. As a consequence, TeX's output routines (and with them mark nodes) were replaced by a template mechanism. Templates, while not as powerful as programs, will always terminate and can be processed efficiently. Whether they offer sufficient flexibility remains to be seen. It is a fact, however, that very few users of TeX or LaTeX write their own output routines. So it can be expected that a collection of good templates will serve most authors well.

The current template mechanism of HINT is still experimental. It is sufficient to replace the output routines of plain TeX and LaTeX.

HINT files contain all necessary resources, notably fonts and images, making them completely self-contained. Embedding fonts makes HINT files larger — the effect is more pronounced for short texts and less significant for large books — but it makes HINT files independent of local resources and of local character encodings. Indeed, a HINT file does not encode characters, it encodes glyphs. While HINT files use the UTF-8 encoding scheme, it is possible to assign arbitrary numbers to the glyphs as long as the assignment in the font matches the assignment in the text. The only reason not to depart from the standard UTF-8 encoding is to maximize compatibility with other software, e.g., to search for user-entered strings or for text to speech translation.

Zoom and size changes

On mobile devices it is quite common to switch within one application between landscape or portrait mode to use the screen space as efficiently as possible. Further, users usually can adjust the size of displayed content by zooming in or out.

For rendering a HINT file, these operations simply translate into a change of `hsize` and `vsize`, with consequences for line and page breaking. While changing line breaks affects only individual paragraphs, changing a page break has global implications, making precomputing page breaks impractical. Consequently, the HINT file format must support rendering either the next page or the previous page based solely on the top or bottom position of the current page. In turn, this implies that it must be possible to parse the content of a HINT file in both forward and backward directions.

A HINT file encodes TeX's contribution list in its content section. To support bidirectional parsing, each encoding of a node starts with a tag byte and ends with that same tag byte. From the tag byte, the layout of the encoding can be derived. So decoding in the backward direction is as simple as decoding in the forward direction.

Changes in TeX's parameters, for example paragraph indentation or baseline spacing, pose another problem for bidirectional parsing. HINT solves this problem by using a stateless encoding of content. All parameters are assigned a permanent default value. To specify these defaults, HINT files have a definition section. Any content node that needs a deviation from the default values must specify the new values locally. To make local changes efficient, nodes in the content section can reference suitable predefined lists of parameter values specified again in the definition section, described next.

Simple and compact representation

At the top level, a HINT file is a sequence of sections. To locate each section in the file, the first section of a HINT file is the directory section: a sequence of entries which specify the location and size of each section. The first entry in the directory section, the root entry, describes the directory section itself. The HINT file format supports compressed sections according to the `zlib` specification [1]. Using the directory, access to any section is possible without reading the entire file.

The directory section is preceded by a banner line: It starts with the four byte word `hint` and the version number; it ends with a line-feed character. The directory section is followed by two mandatory sections: the definition section and the content section. All further sections, containing fonts, images, or any other data, are optional. The size of a section must be less than or equal to 2^{32} bytes. This restriction is strictly necessary only for the content section. It sets a limit of about 500 000 pages and ensures that positions inside the content section can be expressed as 32-bit numbers.

For debugging, the specification of a HINT file also describes a “long” file format. This long file format is a pure ASCII format designed to be as readable as possible. Two programs, `stretch` and `shrink`, convert the short format to the long format and back. They are literate programs [5], and constitute the format specification [10].

Since large parts of a typical content section contain mostly character sequences, there is a special node type, called a text node, optimized for the representation of plain text. It breaks with two conventions that otherwise are true for any other node: The content of a text node cannot be parsed in the backward direction, and it depends on a state variable, the current font. To mitigate the requirement for forward parsing, the size of a text node is stored right before the final tag byte. This enables a parser to move from the final tag byte directly to the beginning of the text. Since text nodes cannot span multiple paragraphs, they are usually short.

Inside a text node, all UTF-8 codes in the range $2^5 + 1$ to 2^{20} encode a character in the current font; codes from `0x00` to `0x20` and `0xF8` to `0xFF` are used as control codes. Some of these are reserved as shorthand notation for frequent nodes. For example, the space character `0x20` encodes the interword glue, and others introduce font changes or mark the start of a node given in its regular encoding.

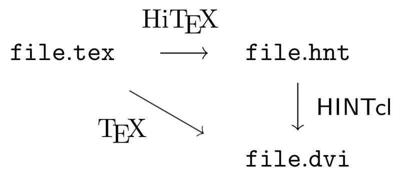
The two forms of content encoding, as regular nodes or inside a text node, introduce a new requirement: when decoding starts at a given position, it must be possible to decide whether to decode a regular node, a UTF-8 code, or a control code. Control codes have only a limited range and the values of tag bytes can be chosen to avoid that range. Conflicts between UTF-8 codes and tag bytes cannot be avoided, hence positions inside text nodes are restricted to control codes. A position of an arbitrary character inside a text node can still be encoded because there is a control code to encode characters (with a small overhead).

Clear syntax and semantics

Today, there are many good formal methods to specify a file format, and the time when file formats were implicit in the programs that would read or write these files seems like ancient history. The specification of the HINT file format, however, is given as two literate programs: `stretch` and `shrink`. The first reads a HINT file and translates it to the “long” format and the second goes the opposite direction and writes a HINT file.

Of course, these programs use modern means such as regular expressions and grammar rules to describe input and output and are, to a large extent, generated from the formal description using `lex` and `yacc`. For this purpose, the `cweb` system [6] for literate programming had to be extended to generate and typeset `lex` and `yacc` files. I consider this representation an experiment. I tried to combine the advantages of a formal syntax specification with the less formal exposition of programs that illustrate the reading and writing process and can serve as reference implementations. The programs `stretch` and `shrink` can also be used to verify that HINT files conform to the format specification.

Specifying semantics is a difficult task and a formal specification is entirely impossible if the correctness depends partly on personal taste. Fortunately the new file format is just an “intermediate” format as part of the TeX universe. So the following commutative diagram is an approximation to a formal specification.



The programs HiTEX and HINTcl mentioned in the diagram are currently in development. HiTEX is a modified version of TEX that produces HINT files as output; HINTcl is a command line program which reproduces TEX's page descriptions as if the parameter `\tracingoutput` were enabled. While it does not actually produce a DVI file, its output can be compared to the page descriptions in TEX's `.log` file to make sure the diagram above would indeed be commutative. The prototypes available so far do not yet support all the features of TEX or HINT.

Conclusion

The experimental HINT file format proves that file formats supporting efficient, high quality rendering of TEX output on electronic paper of variable size are possible. The upcoming prototypes for a TEX version (HiTEX) that produces such files and viewer programs on Windows and Android will provide a test environment to investigate and improve concepts and performance in practice.

In the long run, I hope that a new standard for electronic documents will emerge that enjoys widespread use, has the output quality of real books, is easy to use and powerful enough to encode TEX output, offers the author maximum control over the presentation of her or his work, and can cope with the variations in screen size and screen resolution of modern mobile devices.

References

[1] P. Deutsch and J.-L. Gailly. Zlib compressed data format specification version 3.3. Technical report, RFC Editor, 1996. `tools.ietf.org/html/rfc1950`

[2] EPUB 3 Community Group. epub 3. `w3.org/publishing/groups/epub3-cg`

[3] D. Fuchs. The format of TEX's DVI files. *TUGboat* 3(2):14–19, Oct. 1982. `tug.org/TUGboat/tb03-2/tb06software.pdf`

[4] H. Hagen. Beyond the bounds of paper and within the bounds of screens; the perfect match of TEX and Acrobat. In *Proceedings of the Ninth European TEX Conference*, vol. 15a of *MAPS*, pp. 181–196. Elsevier Science, Sept. 1995. `ntg.nl/maps/15a/09.pdf`

[5] D. E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Center for the Study of Language and Information, Stanford, CA, 1992.

[6] D. E. Knuth and S. Levy. *The CWEB System of Structured Documentation*. Addison Wesley, 1994. `ctan.org/pkg/cweb`

[7] T. Rokicki. *Dvips: A DVI-to-PostScript translator*. `tug.org/dvips`

[8] M. Ruckert. Computer Modern Roman fonts for ebooks. *TUGboat* 37(3):277–280, 2017. `tug.org/TUGboat/tb37-3/tb117ruckert.pdf`

[9] M. Ruckert. HINT: Reflowing TEX output. *TUGboat* 39(3):217–223, 2019. `tug.org/TUGboat/tb39-3/tb123ruckert-hint.pdf`

[10] M. Ruckert. *HINT: The File Format*. Aug. 2019. ISBN 978-1079481594.

◇ Martin Ruckert
Hochschule München
Lothstrasse 64
80336 München
Germany
`ruckert (at) cs dot hm dot edu`