

TUGBOAT

Volume 41, Number 3 / 2020

General Delivery	259	From the president / <i>Boris Veytsman</i>
	260	Editorial comments / <i>Barbara Beeton</i> Passings: Janusz Nowacki, Ed Benguiat, Ron Graham; A new Unicode-specific area in CTAN; Fonts, fonts, fonts; Erratum: “The road to Noto”, Steven Matteson; Learning L ^A T _E X; Graphical history in action; Followup to “old news”; Making <i>TUGboat</i> more accessible
	263	How T _E X changed my life / <i>Michael Barr</i>
Typography	265	Typographers’ Inn / <i>Peter Flynn</i>
Fonts	269	Eye charts in focus: The magic of optotypes / <i>Lorrie Frear</i>
Multilingual	275	The Non-Latin scripts & typography / <i>Kamal Mansour</i>
Document Processing		
Tutorials	281	Using DocStrip for multiple document variants / <i>Matthew Leingang</i>
L^AT_EX	286	L ^A T _E X news, issue 32, October 2020 / <i>L^AT_EX Project Team</i>
	292	L ^A T _E X Tagged PDF—A blueprint for a large project / <i>Frank Mittelbach, Chris Rowley</i>
	299	Functions and <code>expl3</code> / <i>Enrico Gregorio</i>
	308	<code>bib2gls</code> : selection, cross-references and locations / <i>Nicola Talbot</i>
	318	Making Markdown into a microwave meal / <i>Vít Novotný</i>
Graphics	320	User-defined Type 3 fonts in LuaT _E X / <i>Hans Hagen</i>
	324	Data display, plots and graphs / <i>Peter Wilson</i>
Software & Tools	327	Short report on the state of LuaT _E X, 2020 / <i>Luigi Scarso</i>
	329	Distinguishing 8-bit characters and Japanese characters in (u)pT _E X / <i>Hironori Kitagawa</i>
	335	Keyword scanning / <i>Hans Hagen</i>
	337	Representation of macro parameters / <i>Hans Hagen</i>
	341	T _E Xdoc online—a web interface for serving T _E X documentation / <i>Island of T_EX</i>
	343	MMT _E X: Creating a minimal and modern T _E X distribution for GNU/Linux / <i>Michal Vlasák</i>
	346	UTF-8 installations of T _E X / <i>Igor Liferenko</i>
Macros	348	OpT _E X—A new generation of Plain T _E X / <i>Petr Olšák</i>
Reviews	355	Book reviews: <i>Robert Granjon, letter-cutter</i> , and <i>Granjon’s Flowers</i> , by Hendrik D.L. Vervliet / <i>Charles Bigelow</i>
	358	Book review: <i>Glistering</i> , by Peter Wilson / <i>Boris Veytsman</i>
	360	Historical review of T _E X3 / <i>Peter Flynn</i>
Hints & Tricks	368	The treasure chest / <i>Karl Berry</i>
TUG Business	258	<i>TUGboat</i> editorial information
	258	TUG institutional members
	369	TUG 2021 election
Advertisements	370	T _E X consulting and production services
News	372	Calendar

T_EX Users Group

TUGboat (ISSN 0896-3207) is published by the T_EX Users Group. Web: tug.org/TUGboat.

Individual memberships

2020 dues for individual members are as follows:

- Trial rate for new members: \$30.
- Regular members: \$105.
- Special rate: \$75.

The special rate is available to students, seniors, and citizens of countries with modest economies, as detailed on our web site. Members may also choose to receive *TUGboat* and other benefits electronically, at a discount. All membership options are described at tug.org/join.html.

Membership in the T_EX Users Group is for the calendar year, and includes all issues of *TUGboat* for the year in which membership begins or is renewed, as well as software distributions and other benefits. Individual membership carries with it such rights and responsibilities as voting in TUG elections. All the details are on the TUG web site.

Journal subscriptions

TUGboat subscriptions (non-voting) are available to libraries and other organizations or individuals for whom memberships are either not appropriate or desired. Subscriptions are delivered on a calendar year basis. The subscription rate for 2020 is \$115.

Institutional memberships

Institutional membership is primarily a means of showing continuing interest in and support for T_EX and TUG. It also provides a discounted membership rate, site-wide electronic access, and other benefits. For further information, see tug.org/instmem.html or contact the TUG office.

Trademarks

Many trademarked names appear in the pages of *TUGboat*. If there is any question about whether a name is or is not a trademark, prudence dictates that it should be treated as if it is.

[printing date: November 2020]

Printed in U.S.A.

Board of Directors

Donald Knuth, *Ur Wizard of T_EX-arcana*[†]

Boris Veytsman, *President*^{*}

Arthur Rosendahl*, *Vice President*

Karl Berry*, *Treasurer*

Klaus Höppner*, *Secretary*

Barbara Beeton

Johannes Braams

Kaja Christiansen

Jim Hefferon

Taco Hoekwater

Frank Mittelbach

Ross Moore

Norbert Preining

Will Robertson

Herbert Voß

Raymond Goucher, *Founding Executive Director*[†]

Hermann Zapf (1918–2015), *Wizard of Fonts*

*member of executive committee

[†]honorary

See tug.org/board.html for a roster of all past and present board members, and other official positions.

Addresses

T_EX Users Group

P. O. Box 2311

Portland, OR 97208-2311

U.S.A.

Electronic Mail

General correspondence,
membership, subscriptions:

office@tug.org

Submissions to *TUGboat*,
letters to the Editor:

TUGboat@tug.org

Telephone

+1 503 223-9994

Fax

+1 815 301-3568

Web

tug.org

tug.org/TUGboat

Technical support for
T_EX users:

support@tug.org

Contact the
Board of Directors:

board@tug.org

Copyright © 2020 T_EX Users Group.

Copyright to individual articles within this publication
remains with their authors, so the articles may not
be reproduced, distributed or translated without the
authors' permission.

For the editorial and other material not ascribed to a
particular author, permission is granted to make and
distribute verbatim copies without royalty, in any medium,
provided the copyright notice and this permission notice
are preserved.

Permission is also granted to make, copy and distribute
translations of such editorial material into another
language, except that the T_EX Users Group must approve
translations of this permission notice itself. Lacking such
approval, the original English permission notice must
be included.

The most beautiful thing in the world
is a blank piece of paper.

Ed Benguiat (1979), quoted in
New York Times obituary,
16 October 2020

TUGBOAT

COMMUNICATIONS OF THE TEX USERS GROUP
EDITOR BARBARA BEETON

VOLUME 41, NUMBER 3, 2020
PORTLAND, OREGON, U.S.A.

TUGboat editorial information

This regular issue (Vol. 41, No. 3) is the last issue of the 2020 volume year. The deadline for the first issue in Vol. 42 is March 31, 2021. Contributions are requested.

TUGboat is distributed as a benefit of membership to all current TUG members. It is also available to non-members in printed form through the TUG store (tug.org/store), and online at the *TUGboat* web site (tug.org/TUGboat). Online publication to non-members is delayed for one issue, to give members the benefit of early access.

Submissions to *TUGboat* are reviewed by volunteers and checked by the Editor before publication. However, the authors are assumed to be the experts. Questions regarding content or accuracy should therefore be directed to the authors, with an information copy to the Editor.

TUGboat editorial board

Barbara Beeton, *Editor-in-Chief*

Karl Berry, *Production Manager*

Robin Laakso, *Office Manager*

Boris Veytsman, *Associate Editor, Book Reviews*

TUGboat advertising

For advertising rates and information, including consultant listings, contact the TUG office, or see:

tug.org/TUGboat/advertising.html

tug.org/consultants.html

Submitting items for publication

Proposals and requests for *TUGboat* articles are gratefully received. Please submit contributions by electronic mail to TUGboat@tug.org.

The *TUGboat* style files, for use with plain TeX and L^AT_EX, are available from CTAN and the *TUGboat* web site, and are included in common TeX distributions. We also accept submissions using ConTeXt. Deadlines, templates, tips for authors, and more, is available at tug.org/TUGboat.

Effective with the 2005 volume year, submission of a new manuscript implies permission to publish the article, if accepted, on the *TUGboat* web site, as well as in print. Thus, the physical address you provide in the manuscript will also be available online. If you have any reservations about posting online, please notify the editors at the time of submission and we will be happy to make suitable arrangements.

Other TUG publications

TUG is interested in considering additional manuscripts for publication, such as manuals, instructional materials, documentation, or works on any other topic that might be useful to the TeX community in general.

If you have such items or know of any that you would like considered for publication, please contact the Publications Committee at tug-pub@tug.org.

TUG Institutional Members

TUG institutional members receive a discount on multiple memberships, site-wide electronic access, and other benefits:

tug.org/instmem.html

Thanks to all for their support!

Adobe Inc., *San Jose, California*

American Mathematical Society,
Providence, Rhode Island

Association for Computing
Machinery, *New York, New York*

Aware Software, *Newark, Delaware*

Center for Computing Sciences,
Bowie, Maryland

CSTUG, *Praha, Czech Republic*

Harris Space and Intelligence
Systems, *Melbourne, Florida*

Hindawi Foundation, *London, UK*

Institute for Defense Analyses,
Center for Communications
Research, *Princeton, New Jersey*

Maluhy & Co., *São Paulo, Brazil*

Marquette University,
Milwaukee, Wisconsin

Masaryk University,
Faculty of Informatics,
Brno, Czech Republic

Nagwa Limited, *Windsor, UK*

New York University,
Academic Computing Facility,
New York, New York

Overleaf, *London, UK*

StackExchange,
New York City, New York

Stockholm University,
Department of Mathematics,
Stockholm, Sweden

TeXFolio, *Trivandrum, India*

Université Laval,
Ste-Foy, Québec, Canada

University of Ontario,
Institute of Technology,
Oshawa, Ontario, Canada

University of Oslo,
Institute of Informatics,
Blindern, Oslo, Norway

VTeX UAB, *Vilnius, Lithuania*

From the president

Boris Veytsman

There is an interesting paradox in the history of technology. Early adopters start with the first variants of the innovation. They continue to use them while the world around them deploys newer and slicker versions, so the former look rather old and quaint in comparison—precisely because they have been pioneers in the acceptance of the new ideas. This can be seen in many examples; the quirks of the NTSC broadcasting standard adopted in the United States is one example. For another, I remember my surprise when I first saw the inside of Mission Control at NASA Goddard Space Flight Center: the communication devices with their cloth-covered speakers and mother-of-pearl buttons were reminiscent of 1960s turntables rather than futuristic visions of the latest *Star Trek*. Of course these devices were installed in that era and have never needed an upgrade.

The influence of this paradox can be seen in the place of *T_EX* in the free software community—or rather in the wider free information community. *T_EX* users already exchanged tapes of early implementations when the proprietary character of software was taken for granted by many actors in the field. The subsequent appearance of the Comprehensive *T_EX* Archive Network, CTAN, became the model for such archives as CPAN, CRAN, and others. Our flagship publication, *TUGboat*, started publishing papers about free *T_EX* software decades before the *Journal of Open Source Software* was conceived. However, for many users and activists of the free software community our approach may seem definitely quaint and strange. The fact that *T_EX* has a dual role as a program to typeset the texts *and* a language to encode them does not help here. The need to preserve the integrity of the language and the ability to faithfully typeset old manuscripts led to the rather unusual requirements of the L^AT_EX project public license. I remember heated discussions with some purists insisting that LPPL, and the license of *T_EX* itself, were not free. Fortunately, since the LPPL and *T_EX* license requirements were accepted years ago by the GNU Project and the Debian Free Software Guidelines as free, we can put these discussions to rest.

There is, however, another side to the early adopter paradox. The first versions of an innovation often contain more ideas than the later ones. By streamlining the design, the subsequent generations of engineers strip the “unnecessary” ideas and thoughts. Thus, an innovator seeking inspiration is well-advised to study the early works. This is well

known in the arts, where studying and copying old classics is considered an obligatory part of an education. Science and technology students are less keen to study classics—albeit my advisor, Prof. Nikolay Malomuzh, urged us to read papers by Einstein or Bohr rather than their summaries in textbooks. “When you read a textbook,” he said, “you learn only what its author understood in the original paper.”

The *T_EX* community approach to free software is based on ideas from Don Knuth. As a prolific author and mathematician, Knuth followed the old traditions of mathematics when thinking about intellectual property. The way a theorem belongs to its author is quite different from the way Mickey Mouse belongs to Disney Studios. These ideas might be even more relevant now since the free software approach has become popular outside of the world of software itself. Scientific papers are increasingly available on preprint servers and open access journals. Many publishers and granting agencies require the authors to make both their data and code publicly available. The need to significantly accelerate science due to the COVID-19 pandemic has only accelerated these trends. The free software community becomes a part of a more general free information community. There is an understanding that the old licenses and notions based on the experience of free software, while very important, may be not sufficient for the many new kinds of information. The appearance of innovative ideas such as the Creative Commons licenses attests to this understanding. I wonder whether a re-examination of *T_EX* community practices might be useful in the search for approaches to tackle the new reality of the open information epoch.

Besides providing food for thought about the approaches to the intellectual property, our community is also in the business of providing technical means for the free information movement. Many scientific and technical papers—as well as works of fiction, technical documentation, etc.—are typeset with *T_EX*. As always, there is more to do. I think we should do more to aid free tools in supporting advanced features of (the ubiquitous) PDF documents. We need free tools for creation of accessible PDFs—a technology now being increasingly addressed by *T_EX* developers. A less *T_EX*nical but perhaps equally important problem is the improvement of the free PDF reading software, especially in handling PDF forms. I hope our development fund (tug.org/tc/devfund) can help with incentivizing developers to address these problems.

◊ Boris Veytsman
president (at) tug dot org

Editorial comments

Barbara Beeton

Passings: Janusz Nowacki, Ed Benguiat, Ron Graham

This year has not been kind to font designers.

Janusz Marian Nowacki (9 July 1951–7 June 2020) was an active participant in the e-foundry team that created the Latin Modern family and the \TeX Gyre fonts, and an honorary member of GUST, the Polish \TeX group. He was responsible for the revival of several traditional Polish fonts, including Antykwa Toruńska and Antykwa Półtawskiego, which he implemented using MetaType1. His attraction to \TeX was in service to his principal occupation as a rubber stamp maker and his hobby as a fine art photographer. A more personal remembrance appears on the GUST web site.¹

The prominent U.S. font designer Ephraim Edward “Ed” Benguiat (27 October 1927–15 October 2020) is possibly best known for the font bearing his name—ITC Benguiat,² a decorative serif typeface loosely based on typefaces of the Art Nouveau period. This typeface was released in 1977 by the International Typeface Corporation (ITC), an independent licensing company for type designers which he helped establish, and for which he became vice president. Among his other accomplishments was work on the redesign of the *New York Times* logo; this was more a “cleanup” than a full redesign, and the *Times* obituary quotes Benguiat thus: “My thought was, ‘OK, we’ll change it—but if we change it, nobody will recognize it. So all I did was take it and fix it.’” The obituary contains some good advice for aspiring typographers and is well worth reading.³

Not a font designer, but a co-author with Don Knuth and Oren Patashnik of *Concrete Mathematics*, one of the first books to use the AMS Euler font, Ronald Graham (31 October 1935–6 July 2020) was a President of the American Mathematical Society (1993–1994) and creator of the Erdős number, a measure of the distance from Paul Erdős in the collaboration network of mathematical publication.

R.I.P. Messrs. Nowacki, Benguiat, and Graham.

A new Unicode-specific area in CTAN

The new directory tree CTAN:/macros/unicodetex is meant for macro packages that work with either X \TeX and Lua \TeX , but *not* with ‘traditional’ \TeX engines like \TeX and pdf \TeX .

¹ www.gust.org.pl/news/jmn-obit-en

² www.fonts.com/font/itc/itc-benguiat

³ www.nytimes.com/2020/10/16/business/media/ed-benguiat-dead.html

Macro packages that require Lua \TeX (and work with none of the other engines!) are stored in CTAN:/macros/luatex.

Macro packages that require X \TeX (and work with none of the other engines!) are stored in CTAN:/macros/xetex.

Macro packages that work with any \TeX engine, or only with the ‘traditional’ engines, are stored, as before, in CTAN:/macros outside the directories mentioned above.

So far, the following packages have been relocated to CTAN:/macros/unicodetex/latex: fontsetup, fontspec, lilyglyphs, polyglossia, quran, realscripts, texnegr, unicode-math, xltextra.

Feedback would be most welcome about more packages that should be moved to another location, according to this classification. Please email ctan@ctan.org if you are the author, or a knowledgeable user, of a package that you feel should go to the new CTAN:/macros/unicodetex area, but is still located elsewhere on the archive.

Fonts, fonts, fonts

Font Wars

For about half a millennium before \TeX was created, type was metal, but in the mid-20th century, metal type began to be replaced by phototype—negative images of letters and other symbols on film, through which a light was flashed to record an image on a photosensitive surface. Don Knuth had access to an early laser printer, and realized that images could be represented by a matrix of zeros and ones—a bitmap—and from this idea, METAFONT was born, as a necessary adjunct to \TeX . But this was still a specialized operation, carried out on a large shared computer. When, in the 1980s, personal computers became available, one of the first killer applications was word processing. Soon after, personal-sized laser printers appeared, and the race was on to provide fonts that would allow any user of a PC to create any kind of document their occupation required.

Competition among the manufacturers of PCs and associated software was fierce, and the part of it that dealt with the printing of documents became characterized as the “Font Wars”. This period has been described in a number of places, one of which we celebrate here. Chuck Bigelow, as part of a 2017 symposium on the History of Desktop Publishing sponsored jointly by the IEEE Computer Society and the Computer History Museum in Mountain View, California, tells the story of this period in a two-part article: “The Font Wars”, *IEEE Annals of the History of Computing*, 42:1. January–March

2020, 7–40. Additional notes are provided as a web supplement: history.computer.org/annals/dtp/fw/. TUG plans to publish the entire enterprise in book form next year, with additional material. Since Chuck was a key participant in this saga, this is a true first person account, well researched and lucidly presented. Interspersed with the text are numerous illustrations depicting various methods, old and new, used for defining the shapes of letters.

This same subject has been covered from a different point of view (that of someone active in the printing industry) by Frank Romano in his book *History of Desktop Publishing*, which was reviewed in an earlier issue of *TUGboat* (tug.org/TUGboat/tb41-1/tb127reviews-romano.pdf). This book characterizes the different personalities and points of view of the principals involved in the hardware end of the font wars, and is notable for the presence of some photographs that illuminate the intensity of competition that marked the period.

Computer Modern / Latin Modern

Changing gears, visible differences between Computer Modern and Latin Modern were addressed by a question on the `tex.stackexchange` Q&A site (tex.stackexchange.com/q/48369). CM has been criticized for appearing too thin, especially on screen, but a comparison of copies of Knuth's *The Art of Computer Programming* (*TAOCP*) show it to appear more substantial than other documents produced with (I^AT)EX. An answer to this question notes that a great deal of L^AT_EX material is now by default set with Latin Modern, not the original CM. It attributes the difference in weight to the model used for LM (and also for the Type 1 implementations of CM), which has a lesser value for the METAFONT variable `blacker`, leading to noticeably thinner stems.

Other fonts, and some related software

A productive field for font development is directed toward assisting readers with visual disabilities. Several new fonts and supporting software in this area have been announced on the web.

- Luciole (French for “firefly”)⁴ is a sans serif font intended to “advance research”, supporting almost all European languages and including many Greek and math symbols for scientific notation.
- Lexend⁵ is a “variable” font (or series of variations on an underlying basic sans serif font style) that is intended to change depending on feedback from the reader’s ability to comprehend a

text (measured in “words correct per minute”). Its goal is to improve reading proficiency.

- The Accessible RMarkdown Writer⁶ is not a font, but “a tool that creates documents in various formats based on RMarkdown text”. Designed to help create “scientifically rigorous” complex documents, it is based on the existing tool Markdown, with the ability to add inline R-code, and uses pull-down menus to access symbols. BIBTeX is supported for inclusion of references.

Erratum:

“The Road to Noto”, Steven Matteson (*TUGboat* 41:2, 145–154)

A question was raised regarding the relative antiquity of Anatolian hieroglyphics compared to Egyptian, as stated on page 152: “Anatolian hieroglyphs are at least 4,000 years old, thus predating Egyptian hieroglyphs.”

When Matteson was asked about this, he responded, “Whoops — yes I see looking at the timeline I quickly sketched out for the talk I transposed the words ‘Akkadian’ and ‘Anatolian’. Big mistake on my part and I’m very happy it was pointed out.”

But there’s more to this story.

To begin, “Akkadian” applies to cuneiform, not hieroglyphs. Cuneiform has traditionally been considered slightly older than hieroglyphs, but archaeologists are still digging. And the text here mentions only hieroglyphs, not cuneiform. So, where did “Anatolian” come from?

One of the goals of Noto is to support all language scripts in Unicode; the relevant block (U+14400–U+1467F) is named “Anatolian hieroglyphs”. The initial request for adding this script to Unicode (found in the Unicode archives) was submitted by the UC Berkeley Script Encoding Initiative. This document indicates that the script was used by multiple languages, so a regional rather than a linguistic name was applied. Sadly, no temporal information is included.

Elsewhere, in a Wikipedia reference for Egyptian hieroglyphics (https://en.wikipedia.org/wiki/Egyptian_hieroglyphs), it is stated that

Since the 1990s, [...] discoveries of glyphs at Abydos [Egypt], dated to between 3400 and 3200 BCE, have shed doubt on the classical notion that the Mesopotamian symbol system predates the Egyptian one. However, Egyptian writing appeared suddenly at that time, while Mesopotamia had a long evolutionary

⁴ luciole-vision.com/luciole-en.html

⁵ lexend.com

⁶ www.arowtool.com

history of sign usage in tokens dating back to circa 8000 BCE.

It is tempting to infer that the underlying idea was “in the air”, resulting in relatively contemporaneous development.

Learning L^AT_EX

An introductory manual, *Learning L^AT_EX*, by David Griffiths and Desmond Higham, first published in 1997 with a second edition in 2016, has been highlighted by a commentary in *SIAM News*.⁷ In it, the authors predict “future L^AT_EX breakthroughs, leading up to the release of L^AT_EX3 in 2051.” I think that some of these have already come true.

Returning to the present, the new L^AT_EX “instructional” site, learnlatex.org, has in just a few months progressed from an idea to a full-scale operational reality. Introduced by Joseph Wright in a talk at TUG 2020,⁸ the collection of elementary lessons has already been translated into French, Portuguese, Vietnamese and Spanish, and more translations are underway. Each lesson contains one or more typical examples, which can be run directly from the connected page or modified for a different view; experimentation is encouraged. Access to, and presumably use of, the site has grown to several hundred connections per day.

One of the principal goals of *learnL^AT_EX* is to keep it current, unlike many existing web sites that have been constructed but left to lie fallow (sometimes for years), or books that represent a particular point in time.

Graphical history in action

Not fonts, not T_EX, but an art form that *does* utilize fonts as an integral part of its message and considerable charm, posters for the national parks of the U.S. have been compelling advertisements for the locations they picture. Among the most attractive of the lot are the posters created as part of the Works Progress Administration (WPA), an organization created during the Great Depression as a means of supporting artists, whose talents did not otherwise yield a stable existence.

An article⁹ in the *New York Times* tells the story of a retired dentist, Doug Leen, who has tracked down original posters and reproduced them, and also designed new posters in the distinctive style.

⁷ sinews.siam.org/Details-Page/writing-learning-latex
⁸ tug.org/TUGboat/41-2/tb128carlisle-learnlatex.pdf; video: youtu.be/0qTBtKr-5c0
⁹ www.nytimes.com/2020/08/25/style/ranger-doug-leen-wpa-national-park-posters.html

Illustrations accompanying the article show both original and newly created items; it’s very hard to distinguish which is which. (Will there be anyone, or any reason, to memorialize this year’s disaster? The need is certainly there among artists.)

Followup to “old news”

With the cooperation of the Computer History Museum, ACM has carried out interviews with many recipients of the Turing Award. This includes both Don Knuth and Leslie Lamport. The interviews (and transcripts) are online. Start with the main announcement page for Knuth, at amturing.acm.org/award_winners/knuth_1013846.cfm. On that page are links for all forms of the interview, and an alphabetical listing of award recipients. (Leslie is listed next after Don.) The interviews are long—more than seven hours with Don, and about five and a half with Leslie. (I learned from Don’s interview that his long-time secretary, Phyllis Winkler, was able to read his handwriting, was a top-notch technical typist, and thus a superlative candidate for the first non-DEK T_EX tester.) Put this site on your list of things to turn to when you have a few hours of quiet time available.

TUG maintains an extensive list of Knuth videos at tug.org/interviews/#knuthav as well as links for other people in the T_EX world elsewhere on tug.org/interviews. If you know of anything we’ve missed, please let us know.

Making TUGboat more accessible

No, this isn’t a claim that TUGboat will be easier for someone with a visual disability to read, but starting with the next issue, each article will carry a DOI—a Digital Object Identifier. This unique identifier will broaden digital access to TUGboat content, allowing it to be included in major collections of bibliographic data, starting with that of Crossref, the DOI registrar for material like that published by TUG.

The assigned DOI prefix for TUG is 10.47397. To this will be added the journal, volume, issue, and article identification. For new issues, the DOI will appear below the bottom of the first column on the first page of each item to which a DOI is assigned. Assignment of DOIs to earlier issues will proceed as time permits.

The DOIs will be added to Nelson Beebe’s BIB-T_EX database of TUGboat contents.

◊ Barbara Beeton
[https://tug.org/TUGboat/tugboat\(at\)tugdotorg](https://tug.org/TUGboat/tugboat(at)tugdotorg)

How TeX changed my life

Michael Barr

Abstract

I describe the complicated process of writing math papers, getting them typed, and finally getting them published in the days before TeX and how much things changed after.

1 Introduction

As I look back on a career writing papers that started with my 1962 PhD thesis, I am amazed how much things changed from before Don Knuth's amazing program TeX to after. The process of getting math into type was long, messy, and extremely error-prone. It started with writing a fair copy in longhand that a typist could hope to read, followed by correction, submission, revision, rinse and repeat. At the end of the line, the whole exercise had to be repeated with typesetting, all of which changed with the coming of TeX and L^AT_EX.

2 My thesis

Even in retrospect, it was a mess, a long very detailed computation full of Greek letters and loads of subscripts. I was living at home, we rented a typewriter, my mother, who could type, did most of it (with me dictating for the most part) and I did a little. With most of the Greek letters, we left space and wrote them in with a pen. In a few cases, I was able to improvise something. For example, O, backspace, I would give a reasonable approximation of Φ . Or +, backspace, 0 gave something like \oplus . There were no italics; in point of fact until TeX came along, I was not consciously aware that theorems and the like were always set in italics and that math symbols were also italic. I don't believe journals used different fonts for text and math italics, incidentally.

3 Writing papers

I never made any attempt to publish my thesis, although in fact I published the results in much greater generality and without any computation seven years later. But I did start writing papers. To get a paper into print was a long messy process. First write a fair copy longhand. Give it to a department typist and hope that she (it was always a woman) could understand it. A typescript would emerge and I would have to correct it and give it back to the typist. After getting it corrected, I would have to add the Greek letters, symbols, script letters, etc. by hand.

We did have photocopies, but copies cost ten cents a copy, probably more like a dollar in today's money. Still you made a copy. In an earlier era, the

typist might have typed a mimeograph master. I cannot imagine getting that corrected. Even earlier, the hand-written draft would be the one submitted for publication. I once looked up instructions for authors and discovered that the American Math. Society started requiring typewritten submissions only in the early 1920s.

At any rate you submitted your masterpiece to a journal, which sent it out for refereeing. The referee invariably wanted revisions. Many of them involved shortening the paper, removing details, making it harder to read. This wasn't perverse on their part: mathematics was very expensive to print. The paper would have to be revised, which meant starting the whole process over. Assuming the paper was finally accepted, it then got copy-edited. For mathematics, this meant marking up the theorem-like environments for italic, likewise with all the variables, Greek letters, script letters, Frakturs, and special symbols. Then it was sent to a typesetter, who tried to match the marked-up typescript as well as he could. He had about 250 characters that he could put into his machine and he would look at the paper to decide which ones to load. Any additional ones would have to be inserted by hand at extra cost. While 250 sounds like a lot, it has to include the standard alphanumeric characters, italic, bold, large sized for headers and whatever special characters the author might have chosen. Once he had done his best, proofs would be returned to the author, who had to read it again and find and mark errors. Minor author corrections were permitted but discouraged. And nothing that would change the size of paragraphs was allowed. Finally, publication!

Starting in the mid-1960s there were small improvements in the process. The first innovation was things called typits. A typit was a character (symbol or otherwise) on what looked like a typewriter key. You would hold it against the typewriter ribbon and hit any other key. That would hit the typit, which would impress the character on the page. I remember one typist in my department who had a box of typits on her desk and got very fast about finding the required character. Still the process was slow and error prone.

Next came the IBM Selectric typewriter with its typing ball replacing the typewriter keys. I will not try to describe the Selectric (you can google it), but only mention that you could change the ball to a symbol ball or another—there was likely an italic ball—and use that. Still it was a slow process. Replace the ball, find which character to type and do so, then put back the standard ball. But the whole process of writing a fair copy, getting it typed,

correcting it, really hadn't changed. You didn't have as much hand work and the typesetter's job must have been somewhat easier, but it was still slow and expensive.

I think it was in the summer of 1979 that I got so fed up with the whole process that I and my two older children found a book, vintage 1945, "Teach Yourself to Type" and sat down and learned to type. I then bought a second-hand Selectric typewriter along with a symbol ball, determined to learn how to do my own papers. But I never actually did that. Still learning to type was what I would eventually have to do.

4 The coming of \TeX

In 1979 I started working on a book [TTT] jointly with the late Charles Wells of Case Western University. At first, we exchanged typescripts by mail. Unfortunately, mail between Cleveland and Montreal took a minimum of two weeks. In 1980, we discovered [T & M] and decided on the spot that we would try to use \TeX to do our book. My brother, a computer professional, told me we were nuts until we had access to an implementation we could use. We did it anyway. Most readers of this article will not realize the limitations of \TeX 1. Most importantly, there were no add, multiply, or divide instructions. You could increment or decrement a counter by 1, but that was all. Anything like \LaTeX would have been impossible. Nevertheless we persisted. A big problem was the commutative diagrams. We basically left \TeX mode and drew them as best we could using horizontal arrows fabricated with - signs, vertical arrows drawn with | and diagonals with slash and backslash. Arrowheads were done with <, >, v and ^. I guess I assumed that a publisher might do the diagrams in the old-fashioned way and insert them in the right places.

At some point we discovered computer networks and eventually, with much help from our computer centres, we learned to transmit our files electronically. Charles had bought an Apple II in 1979 and by 1982 I had an IBM PC. We once counted that we had used 9 distinct editing programs.

I actually wrote a program that was originally intended to just remove the \TeX code and print out a more-or-less readable manuscript. But in fact I discovered that, using the wonderful (for the time) abilities of the Epson FX-80 printer, I could give a reasonable interpretation of the \TeX code. I also wrote a font generator to generate special fonts for the nine-pin dot matrix printer. The results weren't pretty, but they were readable. It was all monospaced and unjustified.

But we actually had a great stroke of good fortune. We sent it to Springer-Verlag, who handed it off to the editor Roberto Mineo, who was at that time also a graduate student in computer science at Carnegie-Mellon University. He had discovered a beta version of \LaTeX and used it to do the required formatting and also used the \LaTeX picture mode to code all the diagrams. He printed it out at CMU and Springer printed it directly from his camera-ready text. Unfortunately, the printer he used was not the best quality and the original is not up to Springer's standards. A better version is on my website www.math.mcgill.ca/barr.

After that, PC versions of \TeX appeared and I never used a typist again. Life became so much easier. I could typeset a paper, make the necessary revisions in much less time and with much less effort than the old process of getting it ready for publication had been. I never actually used the Selectric and eventually gave it to one of my students who used it for a few years and then got his own computer and learned \TeX . Now I doubt there is a math journal in the world that will accept a paper not composed in \TeX or, most likely, \LaTeX .

Charles and I did one more book together a little later [CTCS]. We did the final version when we were both on sabbatical at Penn in 1990–91. It is interesting to compare timings. In 1990 on a 6 MHz IBM AT computer the compilation took about an hour. Conversion from .dvi to .hp took about an hour and a half and printing on an HP LaserJet took over an hour. I had occasion to recompile a 20% longer version of the book a few weeks ago and producing a .pdf file took all of 8 seconds.

References

- [TTT] M. Barr and C. Wells, *Toposes, Triples, and Theories*. Springer-Verlag, 1985.
- [CTCS] M. Barr and C. Wells, *Category Theory for Computing Science*. Prentice-Hall International, 1990.
- [T & M] D.E. Knuth, *\TeX and Metafont*. Addison-Wesley, 1979.

◊ Michael Barr
404-865 Plymouth Ave.
Mont-Royal, QC H4P1B2
Canada
[michael.barr \(at\) mcgill.ca](mailto:michael.barr(at)mcgill.ca)
<https://math.mcgill.ca/barr>

Typographers' Inn

Peter Flynn

To print or not to print

For over 500 years we have been surrounded by the idea that the final act of creating text is to print it. Then you can bind it, sell it, lend it, circulate it, or whatever you want, because you have ‘it’ in your hands: a book or pamphlet or leaflet, something tangible.

That idea led to the consolidation of conventions in European publishing and elsewhere, some of which was drawn from the manuscript era, about how documents work.

- The document is made up of rectangular pages, held together to form the book.
- The text starts at the beginning, in the appropriate corner, and progresses, symbol by symbol, until the end.
- Along the way it can be broken into divisions according to some conceptual or logical plan defined by the author, which can be used to guide or inform readers.
- There can be other waypoints or milestones to show readers where they are in the document, and to enable them to tell others how to find some item of interest.
- Once we moved from scrolls to pages, a human desire for order in chaos seems to have engendered some conceptions of how things conventionally look:
 - all the pages should be the same size;
 - they should all look roughly the same, or follow a limited set of patterns;
 - they should normally have the same number of lines per page; even when intruded upon by other material (mathematics, music, figures, tables) the positioning of the remaining lines should be consistent.

This is not just to make them easier to bind, but to make them easier to read, and because the people who printed and published the books eventually wanted their editions to be uniform between themselves, but still distinct from everyone else’s.

Take away the idea of printing, and you are left with the PDF or web page on your screen. It may even look like the printed page, but of course it’s just a bunch of colored dots. Yet we keep most of the features listed above because they’re useful to the readers [5]—or we hope they are.

There is a substantial body of opinion, some backed by research and some not, that you should *not* use PDF format for non-print use (e.g. web ‘pages’) because of the potential for severe usability problems compared with conventional HTML:

PDFs are meant for distributing documents that users will print. They’re optimized for paper sizes, not browser windows or modern device viewports. We often see users get lost in PDFs because the print-oriented view provides only a small glimpse of the content. Users can’t scan and scroll around in a PDF like on a web page. Content is split up across sheets of paper, which is fine for printed documents, but causes severe usability problems online. [7]

Normal practice is to publish in multiple formats anyway, with a growing recommendation for HTML5 with CSS3 Paged Media features [9]. However, the use of PDF is in many cases unavoidable for technical or small-p political reasons, in particular the accuracy obtainable with L^AT_EX which is often unavailable in browsers even with HTML5/CSS3, so we need to consider how we can overcome the legacy problems of print. In particular, whichever format you choose (or are required to use), it is essential to make the document accessible according to the prevailing guidelines in your field.

Page numbers. When the idea of the web and other forms of networked electronic publishing caught on, many academic journals and citation format authorities, accustomed to page number references, had serious concerns, because a web page isn’t a page at all—it’s essentially like an endless scroll, able to hold an entire book or even collection of books, with never a page number to be seen. EPUB books change page numbers every time you zoom in or out for a better fit or font. Citation formats that made page numbers compulsory even came under attack for being old-fashioned by some of those who were by now publishing electronically only. Some formats dug their heels in and insisted on page numbers even for pageless documents. That particular panic is largely past, and many journals now retrofit page numbers from the PDF back into the web version (relatively trivial with T_EX).

Margins. Printed books and journals are bound at the left or right edge, according to writing system, which means the inside margin needs to be more than the outside one, to allow for the curvature of the pages close to the spine when the book is open. Historically the margins were a subject of

great care and attention in book design, both in manuscript and print, exemplified by Tschichold's famous diagram (Figure 1). Most printed documents were traditionally set justified, much easier even in hand-set type than in manuscript, so the idea of the text occupying a rectangle of fixed dimensions on each page was an easy convention to continue.

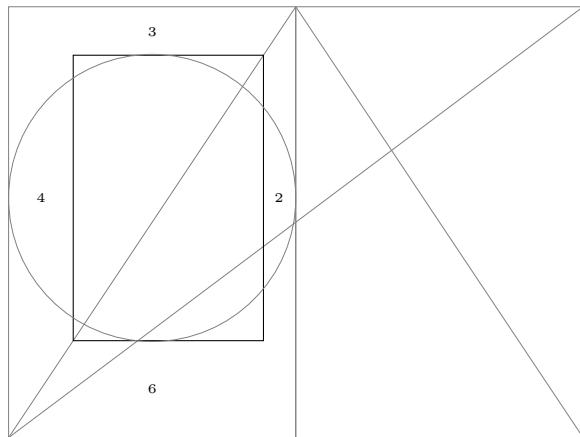


Figure 1: Sketch of page proportions (after Tschichold [8], quoted in Lewis [6]).

In a format designed for on-screen reading, the uneven but symmetrical margins are probably an unnecessary distraction unless you can expect readers to use facing-page software. Some publishers create separate print-ready and display-ready PDFs so that online readers don't see the odd and even margins intended for print.

Lines. While the number of lines per page can be controlled in a PDF, it is pointless and meaningless on the web, and makes an EPUB virtually unusable, as both those formats are designed to be resized by the reader. In any event, line alignment across a double-page spread is not meaningful in a browser or reader unless facing-page viewing is available. The problems of 'show-through', where the aligned or 'backed-up' lines of print on the next or previous page are visible through thin paper, are quite clearly a print-only concept.

Questions. So what should we be looking out for when formatting for non-print reading only? Perhaps the following can act as a starting-point:

- 'Page' shape (window or viewport shape may be a better term): portrait like an office document or landscape like a modern screen?
- Margins: if they no longer need to be asymmetrical, how big should they be?
- Line length: there's more space in landscape, but let's not use it at the expense of readability;

- Font size and leading: how can you use it to compensate for longer lines?
- 'Page' numbering: is it needed at all?
- Number of lines per page or screen or window: is it important?
- Consistency and similarity: do they need to be preserved if more than one document is being published in series?
- Document structure: some form of sectional division will probably continue to be needed; they will require a numbering scheme of some kind if there are no pages to number.

Paper isn't going away any time soon, but as we start to change our reading habits, it's worth starting to think about how that will affect our document classes.

Centering (again)

This has become a recurrent theme as people send me more examples of poor line-breaking in centred titles [2, 3]. In one article [4] I showed an early example (1549) which I reproduce again in Figure 2 where the word 'Contents' was broken 'CON' (in antiqua, large and red, between fleurons) and 'tents' (in blackletter, body text size).

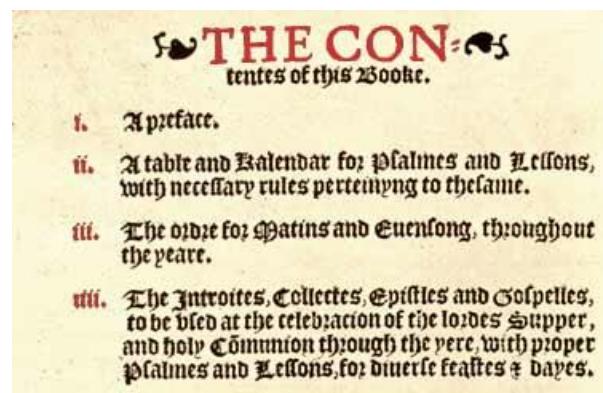


Figure 2: Unusual line-breaking in a heading (Book of Common Prayer, 1549, fragment, courtesy of The Society of Archbishop Justus); see [4] for the original context.

Recently I came across another early example, this time from 1573. It was posted on Twitter as a very low-resolution image on a bright violet background, and I am indebted to Paul W. Nash, Editor of the *Journal of the Printing Historical Society* for identifying it for me, and for providing much additional information (Figure 3). Richard Tottel (also Tottel and other spellings, as here) was a publisher in sixteenth century London, known for his *Miscellany*, the first collection of poetry in English.

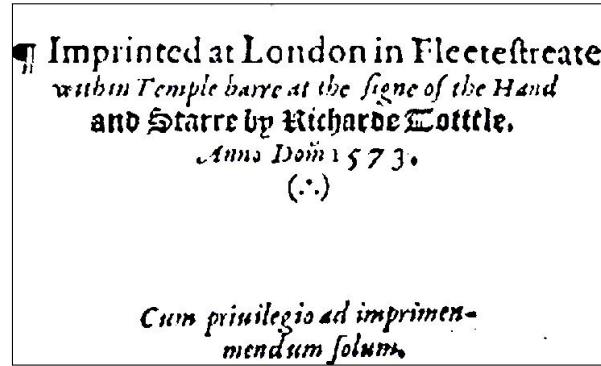


Figure 3: Colophon from Sir William Staunford's *An exposition of the kinges prerogative, collected out of the great Abridgement of Justice Fitzherbert and other olde writers of the lawes of England* (1573) printed by Richard Tottle. Facsimile available at <https://books.google.co.uk/books?vid=0IQ8AAAACAAJ>.

In this colophon, however, it's not a word broken over a line but a phrase: the name of the location (Hand and Star). It is subject to a change of font, again from antiqua to blackletter, which to modern eyes appears strange. But it was a style at the time to alternate lines of different fonts, and Dr Nash is of the opinion that this was the prevailing factor in an arrangement like a colophon where it contributed to successive lines being shorter and shorter to obtain a triangular effect: the relation of the type to the meaning of the text was only considered very loosely if at all. The example is also curious for the extra 't' in the printer's name, and the accidental duplication of the syllable 'men' in the impressum at the bottom.

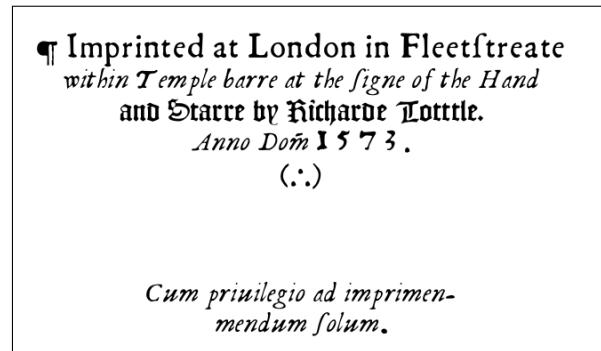


Figure 4: Typographically reconstructed colophon (draft, incomplete)

For a separate project I am using this as an example for typographic reconstruction using easily available modern fonts (Figure 4). In this case the antiqua is Ballard, from Proportional Lime, who specialise in modern cuttings of historical typefaces

(available from MyFonts.com). It is modeled on type used by Henrie Ballard, who ran a press just down the street from Tottle, the other side of Temple Bar, a few decades later. The blackletter is Missaali, a textura based on a much earlier typeface from the German printer Bartholomew Ghotan in the 1480s, and available from CTAN.

Despite the need to retain typographic unity within the line, it is interesting that neither compositors nor printers nor publishers (often the same in those days) felt it necessary for a name or a word to remain in the same font across a line-break. I did at one stage think that perhaps there was a feeling that the publisher's name should be in a specific font, and that there could have been a technical reason behind this—font bodies were not of exact or even sizes between foundries, so a font of a given size from one foundry might not be the same depth as the same font of the same nominal size from another, and would require additional spacing material. But Dr Nash has identified other mixed lines elsewhere in the document which indicate that the smaller size of black letter and italic were indeed cast on the same size of body.

While we're on the subject of mixing fonts, I was sent the sign in Figure 5. At first glance I thought it might be a UK placename like Ottery St Mary or Fornecott St Peter, but apparently it only refers to Osborne Street. Ultimately, if you simply don't have access to the font any more, or it no longer exists in a usable form, your options for changing font in mid-line may be forced.



Figure 5: Garage sign in Colchester, UK

New device driver for old format

I was talking with Barbara Beeton a while ago about a project we are both involved in, and the topic of the durability of text came up. She was making the point that computer files have nowhere near the permanence of clay tablets, which, after all, only become more indestructible when subjected to fire [1].

Given that we can replicate a facsimile of a clay tablet using a 3D printer, and that numerous

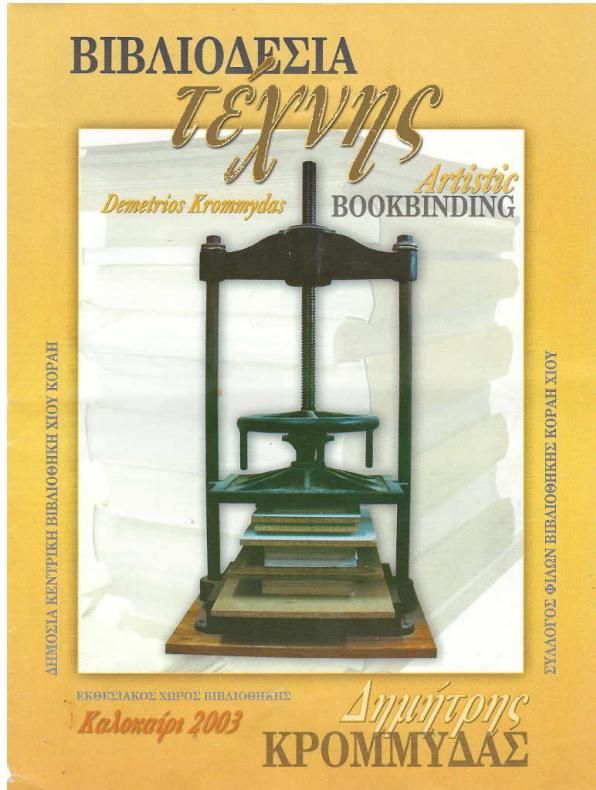


Figure 6: Catalog for *Artistic Bookbinding* (2003) [commemorative for Demetrios Krommydas of Chios (1942–2001)].

cuneiform fonts can be used with L^AT_EX, using the polyglossia package, it should surely be possible to create a dvi2tablet output driver (or an equivalent for a PDF file) so that students worried about the persistence of their dissertation would merely have to translate it into one of the supported languages (Akkadian, Eblaite, Elamite, Hattic, Hittite, Hurrian, Luwian, Sumerian, Urartian, or Old Persian) and output it to a 3D printer, bake the tablets, and store them in a convenient cave.

Afterthought: What's in a name

Anyone who has read documentation about T_EX or L^AT_EX will probably have come across the description of how to pronounce the T_EX bit as ‘tecchh’ because Knuth based it on the Greek $\tau\acute{e}χνη$, meaning ‘craft’ or ‘art’ (as in Knuth’s own *Art of Computer Programming*).

Olivia Fitzpatrick, formerly of UCC’s Boole Library, has shown me a copy of the commemorative catalog for the Greek bookbinder Demetrios Krommydas (Figure 6) which shows the word in a normal Greek context which serendipitously is a craft related to typesetting.

References

- [1] B. Beeton. The future of technical text. In F. Hegland, ed., *The Future of Text*, pp. 58–59. Future Text Publishing, London, Nov 2020.
- [2] P. Flynn. Typographers’ Inn — Titling and centering. *TUGboat* 33(1), May 2012. tug.org/TUGboat/tb33-1/tb103inn.pdf
- [3] P. Flynn. Typographers’ Inn — Afterthought. *TUGboat* 37(3), Sep 2016. tug.org/TUGboat/tb37-3/tb117inn.pdf
- [4] P. Flynn. Typographers’ Inn — Afterthought. *TUGboat* 38(1), May 2017. tug.org/TUGboat/tb38-1/tb118inn.pdf
- [5] P. Flynn. Digital Typography. In K. Norman, J. Kirakowski, eds., *Handbook of Human-Computer Interaction*, pp. 89–108. Wiley, Hoboken, NJ, Jan 2018. <https://doi.org/10.1002/9781118976005>
- [6] J. Lewis. *Typography: basic principles: influences and trends since the 19th century*. Studio Books, London, Jan 1963.
- [7] J. Nielsen, A. Kaley. Avoid PDF for On-Screen Reading. *NN/g Web Usability*, Jun 2020. nngroup.com/articles/avoid-pdf-for-on-screen-reading/
- [8] J. Tschichold. *Designing Books: Planning a book; a typographer’s composition rules; fifty-eight examples by the author*. Wittenborn, Schultz, New York, NY, Jul 1951.
- [9] W3C, Boston, MA. *CSS Paged Media Module Level 3: Working Draft*, Jan 2018. w3.org/TR/css-page-3/

◊ Peter Flynn
Textual Therapy Division,
Silmaril Consultants
Cork, Ireland
Phone: +353 86 824 5333
peter (at) silmaril dot ie
blogs.silmaril.ie/peter

Eye charts in focus: The magic of optotypes

Lorrie Frear

Abstract

An investigation into the optotypes used on eye charts, with comparisons to standard typefaces.

1 Introduction

My graphic design students love to design posters using the classic eye chart composition, and they frequently ask “What typeface should I use for this?” Not having a definitive answer has always been frustrating, so I decided to investigate to find out what typeface is used on eye charts.

I started my quest by asking my ophthalmologist, who enthusiastically provided a dizzying amount of technical information about the variety of eye charts and tests designed for different audiences and eye conditions. Suddenly, a simple question became a series of discoveries. Not only is there *not* one letter-form design or font used for eye charts; the letterform designs are more appropriately called *optotypes*, of which there are several versions. There is a science to the design of optotypes and their legibility at specific distances. Since I am a graphic designer and not an eye or vision expert, I will forgo the technical explanations and focus on optotypes used on several significant charts to provide a better understanding of this complex and fascinating subject.

Eye charts are designed to test visual acuity, or clarity of vision. Each chart design has limitations and advantages, depending on the clinical setting, patient profile, and diagnostic objective. To understand the differences between the charts, it is helpful to know a little historical background of standardized visual acuity testing.

2 The first standardized tests

Heinrich Kühler, a German ophthalmologist, designed a chart in 1836 using figures cut from calendars, books, and newspapers glued in rows of decreasing sizes onto paper. These figures included cannons, guns, birds, farm equipment, camels, and frogs. This system was limited because the figures were not consistent in visual weight or style. Dr. Kühler continued to refine his chart, and in 1843, published a new version using 12 rows of Blackletter letters decreasing in size (fig. 1). This chart was not widely adopted and was published only once in 1843. [6]

The next significant development in visual acuity chart design was the Snellen Eye Chart, which is



Figure 1: Kühler Chart. Heinrich Kühler is one of the first individuals credited with creating an eye chart to test visual acuity.

recognizable to most Americans from visits to the DMV (fig. 2). The Snellen Eye Chart was designed by Dutch ophthalmologist Herman Snellen in 1862 as a means of improving the subjective nature of vision testing, which was usually accomplished by having patients read a passage of text held their hands, or held at a distance by the doctor. This test had obvious limitations: the results were dependent upon the reading ability of the patient, the legibility of the typeface used, and the fact that the patient could guess the next word by reading a sentence. According to Dr. August Colenbrander, a scientist at the Smith-Ketterwell Eye Research Institute and an expert on eye chart design, Snellen began experimenting with dingbats, or symbols such as squares and circles for his eye chart, but found that it was difficult for test subjects to describe the symbols accurately. [5]

So, he moved on to using letters. The characters on the first Snellen Charts were: A, C, E, G, L, N, P, R, T, 5, V, Z, B, D, 4, F, H, K, O, S, 3, U, Y, A, C, E, G and L. The letters used were Egyptian Paragons or slab serifs of contrasting line thickness with ornamental cross strokes on terminals. Snellen then theorized that test subjects would be able to identify non-ornamented, monoline/equally weighted letters of consistent visual size more easily, and so he created optotypes.

At first glance, it may appear that the Snellen optotypes are Lubalin Graph or Rockwell. But upon detailed examination, it is evident that these characters are rather atypical (fig. 3). Unlike typical typefaces in which letter proportions are determined by ‘family’ groupings (such as n, r, m, h and u),

Originally published July 12, 2015 at ilovetypography.com/2015/07/12/what-are-optotypes-eye-charts-fonts.

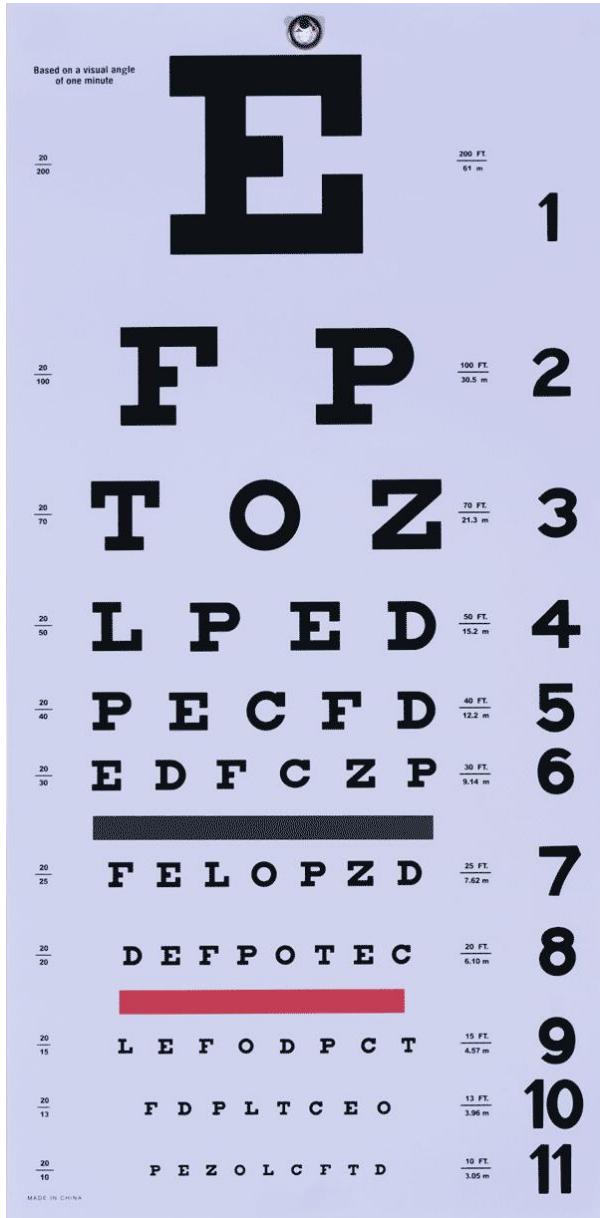


Figure 2: Snellen Chart. (Grayscaled for print; the bar between lines 8 and 9 is red.)

Snellen optotypes are designed on a 5 x 5 grid (fig. 4). Furthermore, they comprise a very limited character set of just 9–10 letters. Optotypes are designed using a simple geometry in which the weight of the lines is equal to the negative space between lines. The height and width of an optotype is five times the thickness of the line weight. [13] These design considerations create inconsistently and oddly proportioned letters. For example, in a typical typeface, C and D would appear wider than Z, but in the optotype scheme, the opposite is true.

Snellen C D Z

Lubalin Graph C D Z

Figure 3: Snellen vs. Lubalin Graph.

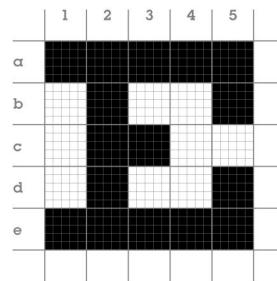


Figure 4: Snellen letter E

Dr. Snellen created optotypes using minutes of arc instead of a typographic measuring system. This made it possible for his charts to be reproduced easily. The first large order for Snellen Charts was from the British Army in 1863. From there, the Snellen Eye Chart became the standard for vision testing for almost a century. In addition, Snellen's 5 x 5 grid optotype design is the foundation upon which all other eye chart systems are based. The Snellen Eye Chart is still the most recognized design, which can, to some extent, negate its effectiveness, if, for example, the test subject has memorized the chart. [5]

Most Snellen Charts contain eleven lines of block letters. The first line consists of a single large letter, most often an E. Subsequent rows have increasing numbers of letters that are progressively smaller in size. The test subject, from a distance of 20 ft, covers one eye, and, beginning at the top, reads aloud the letters in each row. The smallest row that can be read accurately indicates the visual acuity in that particular eye. [9]

Current Snellen Charts use nine letters, C, D, E, F, L, O, P, T, Z. Note that with the exception of E and O, the letters are all consonants. The diverse shapes of the optotypes allow test subjects to identify verticals, horizontals, and diagonals. These letter shapes are also highly effective in identifying astigmatism.

Although today's Snellen Eye Charts may vary in the number of rows, size gradation, and serif or sans serif design [6], their commonalities include

the rectangular shape. This dictates the varying numbers of optotypes appearing on each line as space permits. [8]

As a result of continual refinements, most of today's Snellen Charts follow logarithmic progression, have improved letter designs, and a uniform 25% progression from line to line. [8]

3 Refinements and variations

In 1868, Dr. John Green of the St. Louis College of Physicians and Surgeons in Missouri decided to make some changes to the Snellen Eye Chart. He designed a more structured grid featuring a consistent logarithmic geometric progression of 25% for successive lines, and with proportional spacing. He also changed the style of the optotypes from the blocky slab serif to sans serif. His concept became known as the "Preferred Numbers Series", but his system did not become widely recognized until the next century when sans serif typography gained popularity. Ironically, in response to criticism that his letters looked "unfinished", Dr. Green abandoned them in 1872, and returned to the serif optotypes. [8]

In 1959, Dr. Louise Sloan of Johns Hopkins University created ten new optotypes using sans serif letters preferred by Dr. Green. These optotypes included the letters: C, D, H, K, N, O, R, S, V, and Z. Like Snellen letters, Sloan Letters are formed within a square, with the stroke width equal to one-fifth of the letter height and with equal visual weight. The Sloan Chart (fig. 5) has consistent spacing between letters and rows that are proportional to letter size. Spacing between letters is equal to letter width, and spacing between rows is equal to the height of the letters in the subsequent, smaller row. [4]

Notice that, as in the Snellen Chart, all of the characters are consonants with the exception of O. Also note that the letter selection used on the Snellen Chart is not the same as that in the Sloan Chart. In both cases, the diverse shapes of the optotypes allow test subjects to identify verticals, horizontals and diagonals—an aid to identifying or differentiating individual letters. The ten Sloan Letters are considered to be the most effective letter selection for equal legibility. What's more, they are particularly effective at identifying astigmatism.

The Sloan Letters may at first glance resemble Microgramma or Eurostile (www.myfonts.com/fonts/linotype/eurostile) fonts, but upon closer examination (fig. 6), it is evident again that the grid format imposed upon these optotypes produces some odd and inconsistently proportioned letters.



Figure 5: Sloan Chart.



Figure 6: Sloan letters vs. Eurostile.

4 New charts and methods

In 1976, Ian Bailey and Jan E Lovie-Kitchin of the National Vision Institute of Australia proposed a new chart layout (fig. 7), describing their concept as follows:

"We have designed a series of near vision charts in which the typeface, size progression, size range, number of words per row and spacings were chosen in an endeavor to achieve a standardization of the test task." [11]

This layout replaces the Snellen rectangular chart format with a variable number of letters per

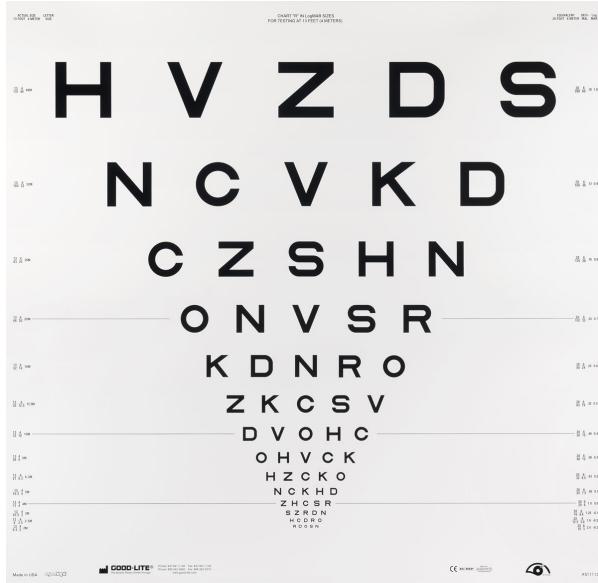


Figure 7: Bailey–Lovie Chart.

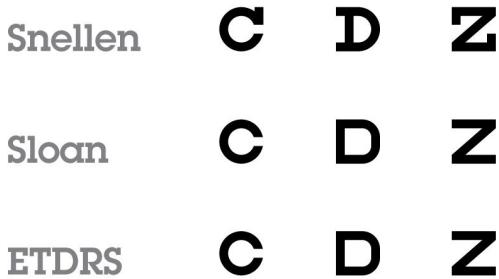


Figure 8: Snellen vs. Sloan letters; Sloan and ETDRS letters are the same.

line with a triangular one with five proportionally spaced letters on each line. The ten Sloan Optotypes appear on the Bailey–Lovie Chart using the same letter ratio of the letter-height equal to five stroke widths, excluding serifs.

The Bailey–Lovie Chart is an example of a LogMAR test, a term describing the geometric notation used to express visual acuity: “Logarithm of the Minimum Angle of Resolution”. Such tests were selected in 1984 as the standard for visual acuity testing by the International Council of Ophthalmology. [3]

In 1982, when the National Eye Institute needed standardized charts for its “Early Treatment of Diabetic Retinopathy Study” (ETDRS), Dr. Rick Ferris combined the Green and Bailey-Lovie Charts’ logarithmic progression and format with the Sloan Letters (fig. 8). ETDRS charts use equal spacing

between letters and lines, making the acuity chart more balanced. [8] This chart format has been accepted by the National Eye Institute and the FDA, and is mandated for many clinical trials performed worldwide.

The ETDRS test is more accurate than either the Snellen or Sloan versions because the rows contain the same number of letters, the rows and letters are equally spaced on a log scale, and individual rows are balanced for letter difficulty. There are also three different versions of the test available to deter memorization. [1]

One limitation of the original ETDRS chart is its use of the Latin alphabet, making it difficult to use throughout all of Europe. To address this limitation, the Tumbling E and Landolt C charts (discussed below) are used for populations who are unfamiliar with letters of the Latin alphabet. Recently, a modified ETDRS chart was developed using Latin, Greek, and Cyrillic alphabets. For this chart, the letters C, D, N, R, S, V and Z have been replaced by the letters E, P, X, B, T, M, and A. These letters are created using the same 5 x 5 grid and the Sloan Letter design. [7]

In more recent years there has been a move to create electronic charts, including the British-designed Test Chart 2000, which was the world’s first Windows-based computerized test chart. It overcomes several difficult issues such as screen contrast, and provides the opportunity to change the letter sequence, so that it cannot be memorized. [2]

These fonts, for Mac and Windows OSs, are available for research purposes. The fonts are based on Louise Sloan’s designs, which has been designated the US standard for acuity testing by the National Academy of Sciences, National Research Council, Committee on Vision. [12]

5 Charts for non-readers

For testing patients who cannot read or for those unfamiliar with the Latin alphabet, the Tumbling E Eye Chart and the Landolt C or Broken Ring Chart are used. [8]

The Tumbling E Chart was designed by Professor Hugh Taylor of the Centre for Eye Research Australia (CERA) in 1978 to test the vision of Australian Aborigine individuals in an attempt to identify those with the eye disorder trachoma.

Professor Taylor, using the Snellen proportions, designed a shape resembling an uppercase E, which he arranged in four directions (up, down, right, and left) in progressively smaller sizes (figs. 9 and 10). The patient then describes the direction in which the Tumbling E is facing.

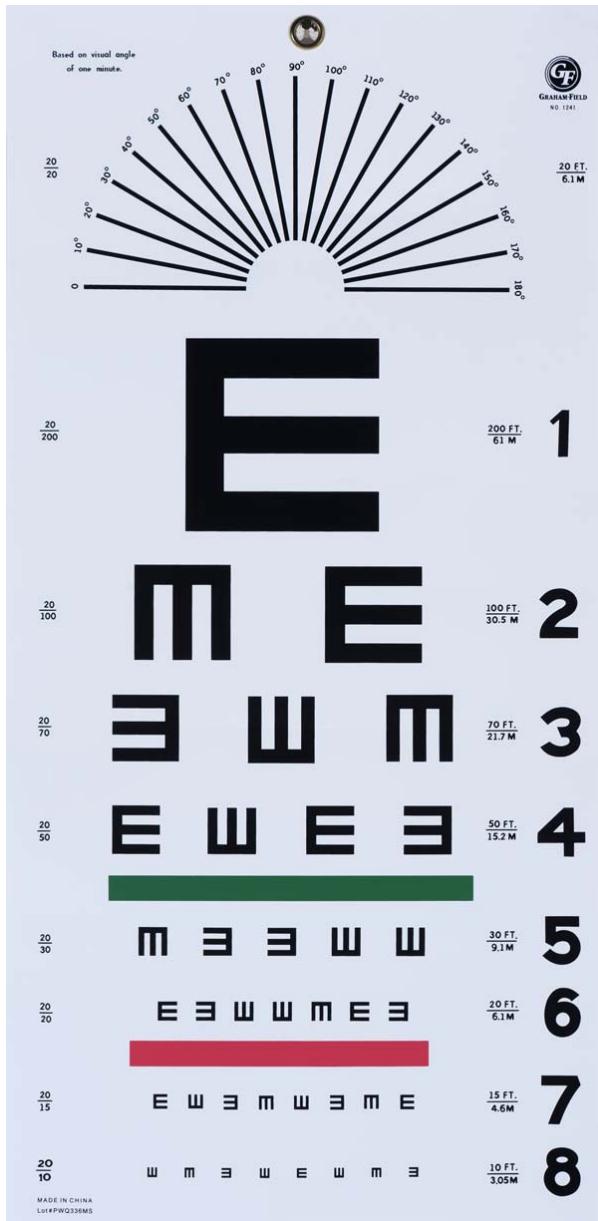


Figure 9: Tumbling E Chart, designed by Hugh Taylor.



Figure 10: Snellen E and Tumbling E.

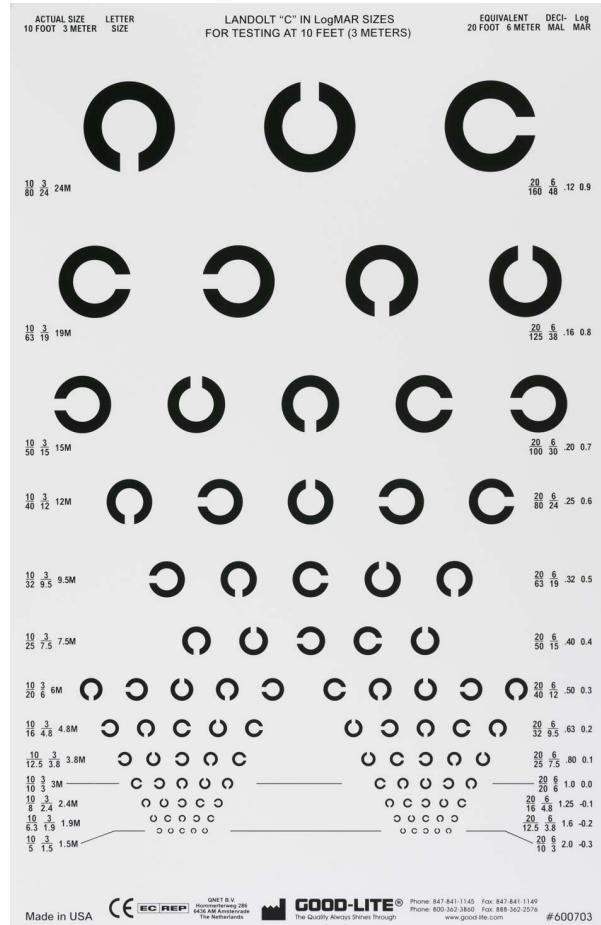


Figure 11: Landolt C (Broken Ring) Chart, designed by Edmund Landolt.



Figure 12: Snellen C and Broken Ring Eye Chart C.

The Landolt C or Broken Ring Eye Chart (see figs. 11 and 12) is also used for illiterate individuals or those persons unfamiliar with the Latin alphabet. Created by Swiss ophthalmologist Edmund Landolt, this test is now considered the European standard. The Broken Ring (which is the same proportions as the C from the Snellen and Sloan Charts) is rotated in increments of 90°. The minimum perceivable angle of the C-gap is the measurement of visual acuity. [10]

In addition to the Tumbling E and Landolt C tests, there are charts for children in which progressively smaller, simple pictograms of objects are used. The challenge in designing these charts is creating

recognizable pictograms of equal visual weight, consistent style, and design.

6 In conclusion

This article is not an exhaustive research study into the subject of eye charts or their efficacy. There are many more examples of eye charts. My objective was to explore the archetypes of optotype design in the evolution of the eye chart as a diagnostic tool. Now I can tell my students that there is, technically, *not* a single typeface to recommend for their designs; and I can refer them to this article for more information! Examining optotypes has been an eye-opening experience.

References

- [1] Bailey, I., Lovie-Kitchin, J. Visual acuity testing. From the laboratory to the clinic. *Vision Research* 20 Sep. 2013; 90:2–9. [sciencedirect.com/science/article/pii/S0042698913001259](https://www.sciencedirect.com/science/article/pii/S0042698913001259)
- [2] The College of Optometrists. Test charts. college-optometrists.org/the-college/museum/online-exhibitions/virtual-ophthalmic-instrument-gallery/test-charts.html
- [3] International Council of Ophthalmology, Visual Functions Committee. Visual Acuity Measurement Standard. icoph.org/dynamic/attachments/resources/icovisualacuity1984.pdf
- [4] Kaiser, P. Prospective Evaluation of Visual Acuity Assessment: A Comparison of Snellen Versus ETDRS Charts in Clinical Practice. *Trans. Am. Ophthalmol. Soc.* Dec. 2009; 107:311–324. www.ncbi.nlm.nih.gov/pmc/articles/PMC2814576
- [5] Kennedy, P. Who Made That Eye Chart? *New York Times*, 24 May 2013. nytimes.com/2013/05/26/magazine/who-made-that-eye-chart.html
- [6] Miranker, E. Just My Optotype. nyamcenterforhistory.org/2017/08/17/just-my-optotype/
- [7] Plainis, S., Moschandreas, J., et al. Validation of a modified ETDRS chart for European-wide use in populations that use the Cyrillic, Latin or Greek alphabet. *Journal of Optometry* Jan.–Mar. 2013; 6(1):18–24. journalofoptometry.org/en-validation-modified-etdrs-chart-for-articulo-S1888429612000817
- [8] Precision Vision. Snellen Eye Chart - a Description and Explanation. precision-vision.com/snellen-eye-chart-a-description-and-explanation/. Also of interest on that site: Measuring Snellen Visual Acuity, precision-vision.com/measuring-snellen-visual-acuity/; Snellen Eye Test Charts Interpretation, precision-vision.com/snellen-eye-test-charts-interpretation/.
- [9] Segre, L. What's an eye test? Eye charts and visual acuity explained. allaboutvision.com/eye-test/
- [10] Wikipedia. Landolt C. en.wikipedia.org/wiki/Landolt_C
- [11] Wikipedia. LogMAR chart. en.wikipedia.org/wiki/LogMAR_chart
- [12] Wikipedia. Sloan letters. en.wikipedia.org/wiki/Sloan_letters
- [13] Wikipedia. Snellen chart. en.wikipedia.org/wiki/Snellen_chart

◊ Lorrie Frear
Rochester Institute of Technology
lorrie dot frear (at) rit dot edu
lorriefrear.info

The Non-Latin scripts & typography

Kamal Mansour

1 Introduction

It is quite common in typographic terminology to divide the world's scripts into Latin and non-Latins. At first glance that might seem to be a reasonable categorization until one looks a bit closer to find that non-Latin consists of a huge number of diverse scripts. Imagine if we were to call Latin script blue, while calling all others non-blue. We would be gathering the remaining colors of the spectrum into one group without differentiation. Calling a color non-blue doesn't tell us anything about what it might be; is it orange, green, magenta, indigo?

However strange this terminology may be, there is good reason behind it. Gutenberg invented moveable type for Latin script; in particular, he focused on a certain Gothic style used at the time by German scribes. We know things didn't stop there. The printed forms of Latin script moved away from the handwritten forms over time, evolving into a distinct craft and discipline. In the process, the letter forms underwent considerable simplification. By the time typographic letters began to develop for other scripts, Latin type had reached a maturity possible only through time. Over the centuries, the machinery and techniques had been refined for the needs of Latin type and any newcomers to the game needed to adapt to the current state of things.

In the realm of non-Latin scripts, what are the main groups and families? Moving eastward from the origins of Latin script, we find Greek script, and further east, its close relative, Cyrillic. Hovering over the Caucasus, Armenian and Georgian raise their heads. Reaching from North Africa eastward, Arabic script occupies a large swath of the landscape that reaches as far as India. Near the juncture of Western Asia and North Africa, Hebrew has its home. In East Africa, Ethiopic script—in ancient times known as Ge'ez—has a sizable presence. Over the large territory of China, as well as in Taiwan, Chinese is the dominant script. Hangul, a syllabic script with some Chinese visual features, dominates in Korea. Meanwhile, further south in Japan, rules a unique hybrid writing system consisting of two syllabaries (katakana and hiragana), Chinese characters, in addition to Latin. The Indian subcontinent is home for a large family of scripts descended from ancient Brahmi writing. Over the centuries, the descendants split into two groups, northern and southern scripts. In the northern group, Devanagari, Bengali, and Gujarati scripts stand out. The southern

group includes Tamil, Telugu, Kannada, Malayalam, and Sinhala. In Southeast Asia, we find more distant descendants of Brahmi in Thai, Lao, Khmer, and Myanmar scripts.

As close relatives of Latin, the Greek and Cyrillic scripts had more time to develop typographically than other non-Latins. In its earliest typographic forms, Greek initially imitated cursive scribal form before eventually settling on printed forms based on its classic origins. Classical and Biblical studies further established the need for Greek typefaces in the late nineteenth and early twentieth centuries.

Cyrillic, originally derived from Greek, was extensively revised and “europeanized” under the hand of Peter the Great. Then early in the twentieth century, Cyrillic was further simplified by purging it of unnecessary vestiges from Greek orthography and redundant characters. In an effort to further unite the huge territory of the Soviet Union, the government extended Cyrillic script to support the many non-Slavic languages of the various republics. Outside the Soviet Union, a certain number of Cyrillic typefaces were always available through the mainstream type foundries to satisfy the needs of academics in Slavic Studies and of emigrant communities.

Because of their common shapes and behavior, Greek and Cyrillic have always been easily accommodated on the same typesetting equipment as Latin script. Of course, the compositor had to be familiar with the script he was setting, with Polytonic Greek being the most demanding because of the variety of diacritics.

When it comes to typography, every script has its intricate details, and consequently, its own story of transition from handwriting to type. It would be interesting to tell every story, but that would be enough to fill many books. Without going into every detail, we can gain some understanding by surveying the types of problems that arose when adapting typesetting technology developed for Latin script to non-Latin scripts. Because scripts have developed as families, they share many attributes with other members of the family. We can take advantage of the similarities to identify recurring obstacles in the typographic development of non-Latin scripts. By citing judiciously chosen examples from a few scripts, we can readily present the gamut of significant difficulties.

Before delving into the range of technical challenges, we must first recognize the most common hurdle shared by all who endeavored to bring a non-Latin script into the typographic age: the fear of the exotic. Typography developed in Europe and it was

Europeans who first strove to take this craft to distant lands. Before the advent of sociology, anthropology, and linguistics, non-European cultures were seen as foreign, exotic, and difficult to fathom. Before understanding the written form of a language, one must first learn the spoken language and, to some extent, understand its culture.

To explore the gamut of typographic challenges, we will visit four scripts: Devanagari, Khmer, Arabic, and Chinese. In terms of visual impression and structure, each of these scripts differs greatly from the others.

2 Devanagari

We begin with Devanagari—the most commonly used script in contemporary India—that is used primarily for the Hindi and Marathi languages. Devanagari has a long history of development that began with the Sanskrit language centuries ago. The structure of Devanagari writing is based on the syllable which, in its simplest form, can be composed of a consonant with either an inherent or explicit vowel mark. As an example, let us consider the letter *ka*, which when written as क, includes the implied vowel *a*. If we want to write *kī*, we need to explicitly add the *ī* vowel ऀ to *ka* क, resulting in की. Presented at a larger size, we can clearly show how the two components of the *kī* syllable overlap on the top:

Now, if we want to write *ku* instead of *kī*, the *u* vowel is placed below *ka* as follows:

Marks other than vowels can also appear above and below letters; for instance, if we want to indicate that the vowel in *ka* sounds nasal, we would add a dot above (natively, *anusvara*) as in:

Since Latin type was usually composed on a single horizontal line with gaps between letters, Devanagari presented quite a challenge because of its need for overlap between adjacent characters, in addition to upper and lower marks.

3 Khmer

Next, we consider Khmer, or Cambodian, writing. As a remote descendant of the Brahmi script, the structure of Khmer is also broadly based on the syllable, including the presence of an implied or explicit vowel; however, the phonetic nature of the Khmer language has also resulted in many additional features that go beyond the ancestral Brahmi model.

The orthographic norms of Khmer script represent the range of simple to complex syllables by allowing its components to stack both horizontally and vertically. Like Devanagari, the simplest syllable consists of a consonant with an implied or explicit vowel, but because of the tonal nature of Khmer, marks indicating tone are sometimes also required. Khmer has the same needs as Devanagari for upper and lower marks, which result in more height and depth for syllable clusters. While Devanagari allows multi-consonant syllables, it also allows them to stack horizontally, if need be. On the other hand, Khmer requires the components of a multi-consonant syllable cluster to stack vertically.

Following is an example of a relatively simple syllable cluster consisting of a single consonant *no* followed by a vowel mark *i* above (u1793 + u17B7):

As an example where additional depth is required, the following syllable cluster consists of a single consonantal letter followed by a subscript consonant that sits to its right while partially overlapping it below [pha + subscript sa]. The use of a subscript form indicates that the preceding consonant is not followed by a vowel (u1795 + u17D2 + u179F):

The character u17D2 indicates that the following consonantal symbol should appear in its subscript form.

There are also deeper cases where a vowel symbol *ie* surrounds and subtends a consonant *ko* followed by a subscript consonant *lo* (u1783 + u17D2 + u179B + u17C0). Note that the right part of *ie* is stretched to embrace the stacked consonants (*ko*+subscript *lo*+*ie*):

The above sequence could also carry an additional mark above it. Unlike Latin letters, Khmer characters have a predominantly vertical aspect ratio when arranged into syllable clusters. One can imagine the numerous challenges posed by Khmer script for those who wanted to implement it in metal type. How does one stack so many vertical elements along the baseline? If Latin and Khmer letters are put side by side, what sort of size ratio should be maintained between the two? Does one resort to using the largest leading needed throughout one document, or does one increase it only when required?

In digital fonts equipped with OpenType technology—or some other shaping software—the built-in shaping logic allows the user to enter text at the character level only, while watching as syllable clusters compose magically into their correct form. By moving from metal to digital type, we also leave the obstacles of metal behind while gaining new flexibility and fluidity of form.

4 Arabic

The cursive form of Arabic script is the one trait that differentiates it from most others. Arabic writing never developed typographic “block” letters, but has remained cursive from its inception to this day.

Spread over many centuries and a broad geographic terrain, many different styles of Arabic script have developed. Under Ottoman rule, when it came time to adapt one of the Arabic styles to type, the Naskh style prevailed. In due time, many other styles were implemented, but Naskh has remained the reference style for running text. Because of its rich variety of contextually variant forms, Naskh style could only be typeset by a compositor well versed in the intricacies of the style. Showing a variety of styles available for manual setting, the following image is taken from an 1873 catalog of the Amiriyah Press in Cairo:



Each of the above styles required that the compositor thoroughly master its shaping rules.

إذا جلسْت تأكلُ معَ مُسلِطِ فَتَأْمَلْ مَا هُوَ آمَانَكَ

Most Arabic text is set without vowel marks, so setting fully vocalized text (with optional marks above and below the letters) as in the above sample was (and still is) laborious, and therefore avoided because of the extra expense.

With the advent of mechanized typesetting in the early to mid-twentieth century, styles had to be simplified to fit the new size constraints of magazines and keyboards. This trend to simplify persisted, reaching its peak with the arrival of phototypesetters where the limited number of character slots was dictated by Latin fonts.

In the present day, one can say with good confidence that Arabic script stands to benefit greatly from digital fonts equipped with OpenType technology in several ways. First of all, the oversimplification of the past century can be discarded because OpenType technology is capable of emulating the various calligraphic styles of Arabic script. Since the global adoption of the Unicode Standard in the 1990s, Arabic text is stored as a sequence of alphabetic characters that are independent of style. The embedded shaping logic of each font reflects its style, while text input remains invariant. By setting a string of text in a particular font, one can expect it to take the shape mandated by the rules of that style. For instance, in the following sample, each line consists of the same text set in three distinct fonts. Note that the second and third lines show words that are set on an incline with respect to the baseline—a calligraphic feature ably rendered in OpenType.

لما كان الاعتراف بالكرامة المتأصلة في جميع أعضاء الأسرة البشرية وحقوقهم المتساوية الثابتة
لما كان الاعتراف بالكرامة المتأصلة في جميع أعضاء الأسرة البشرية وحقوقهم المتساوية الثابتة
لما كان الاعتراف بالكرامة المتأصلة في جميع أعضاء الأسرة البشرية وحقوقهم المتساوية الثابتة

5 Chinese

We now take a look at the fourth script, Chinese, which is nowadays shared mainly by the Chinese and Japanese languages, albeit with some stylistic differences. Unlike the three previously examined scripts, Chinese characters require no additional marks nor any contextual shaping. Once designed, a character can simply be typeset as is. The primary obstacle for Chinese script lies in the thousands of characters needed to set everyday text. Chinese writing has a

...Inherent dignity and...inalienable rights of...the human family is the foundation of freedom
...Inherent dignity and...inalienable rights of...the human family is the foundation of freedom

Figure 1: Typesetting in a standard Latin font vs. Zapfino.

long history of exacting calligraphic styles that require great effort to master. The following sample shows Song, one of the most common styles.

鉴于对人类家庭所有成员的固有尊严及其平等的和不移的权利的承认,

Despite the graphic complexity of its characters, Chinese script is typically classified among the “simple” scripts because the typesetting of Chinese text has no complexities in terms of its character shapes.

To write Japanese, Chinese characters (*kanji*) are intermixed with two phonetic syllabaries called *hiragana* and *katakana* to show grammatical affixes, as well as foreign words and names. In typical Japanese text, about half the characters consist of kana (*hiragana* or *katakana*). The following text is taken from the first line of the Universal Declaration of Human Rights. The more fluid and curvier character are *hiragana*.

人類社会のすべての構成員の固有の尊厳と平等で譲ることのできない権

In the twentieth century, ingenious methods and typesetting equipment were devised to handle the large character sets needed for Chinese and Japanese in their home lands. However, the keyboards and magazines of the mechanical typesetters of Linotype and Monotype were designed for limited character sets. In addition, the cost of developing an adequate set of Chinese characters was high enough to stop any Western companies from entering the field until the early era of digital typesetters in the late 1980s, when Monotype entered the marketplace.

Once digital methods prevailed in the Chinese and Japanese markets, more local enterprises were able to enter the typographic field and thrive. Digital text entry now depends on sophisticated software that exploits the systematic structure of Chinese character in terms of its radicals, strokes, or phonetics to simplify the job for the user.

6 Degrees of complexity

Now that we have taken a quick glimpse at several scripts and their traits, we need to examine what inferences we can draw for the world of scripts.

Scripts vary greatly in their level of complexity. A typical Latin typeface is quite simple because of its relatively small character set and the near absence of shaping rules, while a calligraphic font such as Zapfino depends on a set of complex rules; see fig. 1

This example clarifies how complexity can often be based in a particular style, and is not necessarily fundamental to the structure of the script itself. Some scripts are relatively complex because of their intrinsic shaping rules, while their input requirements are simple due to a small character set. The line composition of Chinese text is relatively simple because all characters fit within a square and require no additional marks. On the other hand, the labor needed to create all the characters is disproportionately high, and input methods are complex by necessity because of the large character set. An Arabic font whose style requires at most four shapes for each letter is simple in comparison to one whose style requires a rich set of contextual variants. Complexity in scripts is a matter of gradation. There is no need to simplistically put all scripts into two heaps, simple or complex.

In this brief overview, we have seen some salient features of each of the following scripts: Devanagari, Khmer, Arabic, and Chinese. Although there are no visual similarities among these scripts, they do share a common history: each of them came into the twentieth century with a typographic tradition of varying durations and strengths. While this is also true of other scripts, there are many others that came into the twenty-first century with either a typographic false start or none at all. There are many language communities who had their own traditional script but whose populations were too small to initiate and sustain a typographic practice. Other communities had established a traditional practice of metal typesetting during the nineteenth or twentieth centuries, but could not transition to newer technology such as phototypesetting as metal technology became obsolete.

Let us now roll back the calendar by about four decades to recount how we got to the current state of typography. As digital technology was maturing in the 1970s and 1980s, a silent software revolution was starting to overtake typography. Donald Knuth’s METAFONT and TeX started making digital

typography directly accessible to academics. This transition enabled a wave of academic publications that had earlier been throttled by ever-rising costs of professional typesetters. In the mid-1980s, the wave of WYSIWYG word-processing began at Xerox, but was later made available to the masses through the Apple Macintosh. The concurrent publishing of the PostScript language, and its use as a standard font format, wrested digital typography from the hands of proprietary typesetting equipment and brought it into the public arena. High-resolution printing—well, 300 dpi at the time—suddenly appeared in big and small offices alike, dragging along with it the word *font* into everyday language.

The first stage of the typographic revolution enabled the definition of character shapes through software instead of hardware. As affordable character-drawing software became available in the early 1990s, more typographic aficionados were drawn to the field as new practitioners. The new software made it possible to produce consistent character sets, properly formatted as fonts destined to create high-resolution imprints on paper. Because of the notable difference in resolution, what appeared on the computer screen was only an approximation of the final image produced by a laser printer on paper. In the end, the paper image was the more important one. After all, until that time, typography's aim was to leave an imprint on the relatively permanent medium of paper.

By making letter forms in software instead of metal, the wall of separation between Latin and non-Latin characters collapsed. After all, outlines defined in the form of polynomials have no physical barriers like molds and grooves, and are even free to overlap and intertwine. The domain of digital character shapes proved itself to be adaptable to any script.

As impressive as the first stage of digital typography was, it was only a first step forward. Even though it allowed us to produce typographic images without resorting to metal, it still had significant deficiencies which were not immediately apparent to all. Once stored on a computer, one could use a text document to produce numerous impressions of it over time. In a sense, we can consider the document a digital equivalent of the typographic galley for metal type, but unlike the galley, the early digital documents were somewhat unreliable. In the early 1990s, typical text documents used inherently ambiguous character codes; for instance, the code 192 (decimal) represented Å in a Latin-1 code set, while it meant ω in a Greek set. With the right software and fonts in place, such a document could produce

a faithful image of the intended content, but it was not a dependable, unambiguous archive of the original text.

Around that time, a group of technologists involved with multilingual computing were discussing how to achieve the until-then elusive goal of an unambiguous, global character set. They wanted a digital environment where code 192 would represent only one character, and every character of every language would be assigned a unique code. Gone would be the world of ambiguous character codes, and all scripts, Latin and non-Latin, would be on equal footing. This group of visionaries laid down the foundation for the Unicode Standard, which eventually came to be recognized as the one global character set by all the major makers of systems and software.

Once Unicode was accepted by Apple and Microsoft, the slow transition away from ambiguous codes began. Before and during this transition, the transfer of text documents between different systems was laborious. Any characters outside the 7-bit ASCII range could be garbled in transit unless they had been deliberately filtered. For instance, code 0xE8 in Mac Roman represented Ą, while in Windows Code Page 1252 it stood for è. As the use of personal computers continued to penetrate every aspect of life, it became clear that such a contradictory situation was untenable in the long run. Change was on the horizon, but it was slow in coming.

In the mid-1990s, as the presence of the Unicode Standard made itself felt across the globe, it met resistance in many different regions because it was perceived as foreign-born. To be accepted everywhere, advocates and practitioners of the Standard had to demonstrate its efficacy and suitability before many different national and linguistic communities. As this defense of the Standard was mounted, it soon became apparent that an aspect of Unicode had turned out to be an unintended obstacle for some non-Latin scripts.

“Characters, not glyphs”—one of Unicode’s primary design principles—was aimed at keeping plain Unicode-encoded text free of the entanglements of style. Characters represent abstract units such as letters of the alphabet, while glyphs depict the particular style and size of the character, as dictated by a specific font. Since Unicode represents only the characters of a string, the final shape and appearance of the display glyphs are relegated to the “text rendering” process.

In the mid-1990s, competing systems had different “rendering” processes that typically supported only a few scripts. The rendering component of systems turned out to be a bottleneck for “complex”

scripts such as Devanagari and Arabic; though their texts could be readily encoded in Unicode, it was difficult to achieve good typographic results. In that period, Apple introduced a sophisticated typographic system called “Apple GX” which was capable of rendering many scripts and styles, but it ran only on Apple hardware. Soon thereafter, Microsoft countered with a competing system called “TrueType Open”.

Every practitioner in the publishing community was faced with a difficult choice to make. It wasn’t until 1996 that Adobe and Microsoft first agreed on OpenType, a font specification that integrated both PostScript Type 1 and TrueType font formats, in addition to a unified system supporting advanced typographic features. Finally, there was hope of producing cross-platform fonts in the near term, even for non-Latin fonts. As could be expected, this potential was achieved gradually over the following years.

Today, we can rightly claim that a large number of scripts can be correctly rendered and that the level of typographic sophistication is continually rising. Certainly, scripts such as Devanagari, Arabic, and Khmer can be correctly rendered on multiple digital platforms without any worry about portability. The use of Unicode for numerous scripts, and the languages they support, has been universally accepted. Each of the characters A , $\bar{\omega}$, \ddot{E} , and \grave{e} is represented by the same unique code in any document, on any platform.

At this auspicious juncture, one must ask if there is more to do. Have we reached a stage where all the scripts in the world are on an even footing? In the Internet era, what does it take for a script to become “digitally enabled”? To prepare text documents in a particular script, all its characters must first be listed in the Unicode Standard, each of them assigned a unique code. Then, to make such documents humanly readable, there must be at least one functional font that renders the text in its commonly recognizable form. Once these two requirements are met, documents in a given script—and all the languages it supports—can be created, read, distributed, and archived.

Even though Version 13.0 of Unicode includes 154 scripts, more than 100 scripts are known to be under consideration for future inclusion. It might come as a surprise to some that so many scripts remain unencoded. Some scripts might no longer be in use and require considerable research to produce a definitive character set, while others might belong to a minority population that does not have the resources to prepare the documents needed to propose their inclusion in the Standard. Since it was estab-

lished in 2002, the Script Encoding Initiative (SEI) has been preparing and presenting such proposals for scores of less privileged scripts (see linguistics.berkeley.edu/sei/scripts-encoded.html). Over the years, SEI has been instrumental in bringing at least 70 scripts into Unicode. Without such sponsorship, many scripts have little chance to enter the digital era.

About 10 years ago, Google launched the Noto Project (google.com/get/noto) to build OpenType fonts for the scripts in Unicode, and to make the fonts freely available to all under an open source license. Noto fonts were required not only to support the character set for each script, but also to make use of advanced typographic features that render the script in its expected native appearance. So far, at least 120 scripts are supported by Noto fonts. For scripts that were already widely covered, the Noto fonts simply broadened the variety of styles available, while for other scripts, they opened the door to the digital domain.

Not all scripts are on even footing yet—a great many have just begun their “digital lives”. As always, improvements will need to be made, and additional scripts will have to be encoded. All the same, I can say with confidence that this is a fruitful and auspicious moment for the whole spectrum of global scripts, from red to violet.

◊ Kamal Mansour
 Kamal.Mansour (at) {monotype,me} dot com

Production notes

Karl Berry

This article was typeset with X_ET_EX, using the `polyglossia` package and its commands `\texthindi` and `\textkhmer` for the Devanagari and Khmer examples. The input text was UTF-8.

We used the Noto Serif Devanagari and Noto Serif Khmer fonts (regular weight). It was not feasible or desirable to install them as system fonts, so we specified them to `polyglossia` as filenames:

```
\setotherlanguages{hindi,khmer}
\newfontfamily\devanagarifont[Script=Devanagari]
  {NotoSerifDevanagari-Regular.ttf}
\newfontfamily\khmerfont[Script=Khmer]
  {NotoSerifKhmer-Regular.ttf}
```

With the addition of `,Renderer=HarfBuzz` in the `\newfontfamily` calls, the results obtained with L_UA_TE_X were identical. ◊

Using DocStrip for multiple document variants

Matthew Leingang*

Abstract

I describe a method of keeping multiple variants of the same document within a single file, using DocStrip.

1 Introduction

As a college professor, there are several times when I need to keep teaching materials in several different forms. For example:

- A single class day’s lesson might consist of lecture slides in the `beamer` class, a handout of the same slides printed 2–3 on a page for student notes, my own lecture notes as a manuscript, a worksheet for in-class activity, and solutions to that worksheet.
- A single week’s homework assignment might consist of problem statements, with hints and reading notes, a L^AT_EX template for students to fill in with their own answers, and solutions with comments to be published after the assignment has been graded.

Over the years I have developed a workflow for maintaining these “bundles” of documents in the same file, using the L^AT_EX DocStrip utility. This workflow allows me to programmatically vary the content and formatting, and avoids external scripts or filesystem hacks. In this article I will introduce the reader to DocStrip and explain how I use it.

2 The DocStrip utility

DocStrip [5] was originally designed as a literate programming method. L^AT_EX package and class authors use it to write documentation for their code in the same file as the implementation, within commented lines. DocStrip would *strip* out the comment lines and use them to produce *documentation*. The slimmer package and class files would be installed to save compile time. In subsequent versions DocStrip developed the ability to write lines to several different files in one batch, through the use of *options*.

DocStrip batches are programmed in a T_EX file as in Listing 1. The `\input` line loads the DocStrip code. The `\generate` line instructs T_EX, effectively, to “read `foo.dtx` and write `foo.sty`, setting the `package` option”.

When generating files, DocStrip ignores all lines beginning only with a single %. Non-commented lines

* The author wishes to thank the editors and reviewers for their thoughtful and productive feedback.

Listing 1: A minimal DocStrip batch file

```
\input docstrip.tex
\generate{\file{foo.sty}
  {\from{foo.dtx}{package}}}
\endbatchfile
```

are written to all destination files. Lines beginning with a *guard* will be written to the destination file depending on the options set. Each guard begins with a % and contains a boolean expression enclosed by angle brackets. For example, a line beginning with %<bar> will only be passed to generated files when the `bar` option is set. DocStrip can generate more than T_EX files, too; for example, BIBT_EX files, data files, and shell scripts can be embedded in the master file and extracted.

Now putting guards at the start of every line would be cumbersome to type. So guard modifiers are used to delimit blocks of code with the same guard. Any expression preceded by '*' will apply the indicated guard to every line that follows, until the identical expression is encountered with the '/' modifier. This gives an almost HTML-like layer to the DocStrip source file, where blocks of code between lines starting with %<*bar> and %</bar> will be written to any file generated with the `bar` option.

A DocStrip batch declaration such as in Listing 1 often resides in a separate file. If this code were in `foo.ins`, running T_EX on `foo.ins` would extract `foo.sty` from `foo.dtx` and quit. But DocStrip files can be also be configured to “self-extract” by putting the batch declaration at the beginning of the file. In this configuration, running T_EX on `foo.dtx` will instruct T_EX to parse `foo.dtx` a second time, this time writing `foo.sty`. Thus, the document content and extraction instructions can reside in a single file.

3 Example: A problem set with answer template and solutions

As a small but not quite minimal example, let’s consider a DocStrip file called `hw.dtx`. The entire file can be viewed online as part of the github repository for this article: github.com/leingang/tugboat-docstrip.

3.1 Batch header

Listing 2 shows the beginning of `hw.dtx`, which loads `docstrip.tex` and declares the `\generate` batch. It’s surrounded with `driver` guards. Since no generated file sets the `driver` option, this block is not written to *any* file.

We generate four files:

Listing 2: The header block of `hw.dtx`, declaring `\generate` batch

```

1 %<*driver>
2 \input docstrip.tex
3 \askforoverwritefalse
4 \generate{
5   \file{\jobname.qns.tex}{\from{\jobname.dtx}{questions}}
6   \file{\jobname.ans.tex}{\from{\jobname.dtx}{questions,answers}}
7   \file{\jobname.sol.tex}{\from{\jobname.dtx}{questions,solutions}}
8   \file{\jobname.bib}{\from{\jobname.dtx}{bib}}
9 }
10 \endbatchfile
11 %</driver>

```

- `hw.qns.tex`, which sets the option `questions`. This document will be the prepared questions sheet for the instructor to distribute to the students.
- `hw.ans.tex`, which sets the options `questions` and `answers`. This file will be distributed to students as L^AT_EX source, so that they can fill in their answers without having to create their own file from scratch.
- `hw.sol.tex`, which sets the options `questions` and `solutions`. This can be published once the assignment is collected and graded.
- `hw.bib`, which only sets the `bib` option. This is a BIBL^EX file that can be included in any of the L^AT_EX files. If the assignment needs to be copied with only a few changes, such as the year and the due date, only one file must be copied from the old directory to the new.

Lines 12–89 of `hw.dtx` are delimited with the `<questions>` guards and enclose an entire L^AT_EX document from `\documentclass{article}` through to `\end{document}`. Lines 90 and onward (not shown in this article) are delimited with `<bib>` and comprise the complete `hw.bib` file.

3.2 Using guards to conditionally include text

Listing 3 (following page) shows an excerpt of `hw.dtx` that declares a question. Notice that the `hint` environment is surrounded by a guard with a compound boolean expression `<!answers&!solutions>`. The effect is that the `hint` is shown when the `questions` option is selected, but `answers` and `solutions` are *not* selected; that is, only in the `hw.qns.tex` file. The resulting block that is written to the questions file is shown in Listing 4.

Comment lines that begin with a single % are stripped from the input and do not get printed to any output file. But comment lines beginning with two % characters remain. So the comment on line 71

of `hw.dtx` is retained in the `hw.ans.tex` file (Listing 5). It is a note to the student where to write their answer. Finally, the `solution` environment and subsequent commentary paragraph are written to the `hw.sol.tex` file (Listing 6).

3.3 Using guards to conditionally define environments

The implementation of the environments `question`, `answer`, `hint`, and `solutions` have to be set up in the preambles of the generated L^AT_EX files (or in packages used by them). But guards can be used in the preambles too. In this way, we can conditionally style the document.

For instance, I prefer that the question text be upright in the questions file and italicized in the answers/solutions file. This is accomplished in Listing 7. Line 34 is written to the answers and solutions file, and overrides line 33. The preamble of the questions file defines `question` under the `definition` theorem style, with bold header and upright body font. But in the answers and solutions file, the `plain` theorem style is in force, so `question` sets its body in italic.

Listing 7: An excerpt of `hw.dtx` (lines 35–39) showing conditional styling of the `question` environment

```

\usepackage{amsthm}
\usepackage{amssymb}
\theoremstyle{definition}
%<answers/solutions>\theoremstyle{plain}
\newtheorem{question}{Question}

```

3.4 Advanced tricks

If you look in the full `hw.dtx` file online, you'll see a few more automatic variations with DocStrip:

- The document title is specified in `hw.qns.tex`. In `hw.ans.tex`, the text ‘Answers to’ is prepended to the title (using the `\preto` command from the `etoolbox` package). In `hw.sol.tex`, the phrase ‘Solutions to’ is prepended. In

Listing 3: An excerpt of `hw.dtx` declaring a question (the question is from [7])

```

61 \begin{question}
62   \cite[Exercise 6.6]{Scheinerman}.
63   Disprove: if  $p$  is prime, then  $2^p - 1$  is also prime.
64 \end{question}
65 %<!*answers&!solutions>
66 \begin{hint}
67   All you need is one counterexample. Guess and check, and be persistent.
68 \end{hint}
69 %</!answers&!solutions>
70 %<*answers>
71 %% Student: put your answer between the next two lines.
72 \begin{answer}
73 \end{answer}
74 %</answers>
75 %<*solutions>
76 \begin{solution}
77   Let  $p=11$ . Then  $p$  is prime. But  $2^{11}-1 = 2047 = 23 \times 89$ .
78   So the statement is false.
79 \end{solution}
80 A prime number that is equal to  $2^n-1$  for some  $n$  is called a \emph{Mersenne}
81 Prime}. Examples of Mersenne primes are  $3 = 2^2 - 1$  and  $127 = 2^7 - 1$ . It is
82 unknown whether the number of Mersenne primes is finite!
83 %</solutions>

```

Listing 4: The generated block in `hw.qns.tex`

```

61 \begin{question}
62   \cite[Exercise 6.6]{Scheinerman}.
63   Disprove: if  $p$  is prime, then  $2^p - 1$  is also prime.
64 \end{question}
65 \begin{hint}
66   All you need is one counterexample. Guess and check, and be persistent.
67 \end{hint}

```

this way, the original title only needs to be put in one place.

- The document author is set differently in the different document variants. In `hw.qns.tex` and `hw.sol.tex`, the author is the professor. In `hw.ans.tex`, the author is set to a generic student name, and `\LaTeXWarning` is used to remind the student to change the generic name to their own name.

4 Comparing alternatives

The problem of maintaining the sources for different, but closely related documents in the same file, and specifying which documents are to be typeset at the time of compilation, has been encountered by users before, for instance on the Stack Exchange network [1, 4, 6]. Let's consider some of the alternatives in the context of the example use case from Section 3.

4.1 Separate files

In this strawman workflow, separate, nearly identical files are kept side-by-side. Any correction (to a problem, for instance) requires three files to be updated. Further, copying a question to another assignment requires copying from three different files to three different files. This setup is ripe for inconsistency and headache.

4.2 Option setting in pure (I^A) \TeX

In this workflow, certain (I^A) \TeX booleans are defined and set, and code is conditionally executed depending on those booleans. The booleans can be set by adding \TeX code on the command line as optional arguments to the executable (e.g., `pdflatex`). Or, a shell file can be created which sets options, then inputs a common master file.

When options are set within the document, the `\jobname` is the same independent of the options

Listing 5: The generated block in `hw.ans.tex`

```

50 \begin{question}
51   \cite[Exercise 6.6]{Scheinerman}.
52   Disprove: if  $p$  is prime, then  $2^p - 1$  is also prime.
53 \end{question}
54 %% Student: put your answer between the next two lines.
55 \begin{answer}
56 \end{answer}

```

Listing 6: The generated block in `hw.sol.tex`

```

57 \begin{question}
58   \cite[Exercise 6.6]{Scheinerman}.
59   Disprove: if  $p$  is prime, then  $2^p - 1$  is also prime.
60 \end{question}
61 \begin{solution}
62   Let  $p=11$ . Then  $p$  is prime. But  $2^{p-1} = 2^{11-1} = 2047 = 23 \times 89$ .
63   So the statement is false.
64 \end{solution}
65 A prime number that is equal to  $2^n - 1$  for some  $n$  is called a \emph{Mersenne}
66 Prime}. Examples of Mersenne primes are  $3 = 2^2 - 1$  and  $127 = 2^7 - 1$ . It is
67 unknown whether the number of Mersenne primes is finite!

```

set. In our homework example, this would mean the problems file and solutions file would both end up named `hw.pdf`. Not only does this mean that the problem set and solutions PDFs cannot inhabit the same directory at the same time, one can be mistaken for another. Imagine the poor professor distributing what he thought was the questions-only PDF, only to realize that he had instead shared all the solutions!

With shell files, the `\jobname` is different for each document variant, avoiding the possibility of such a mistake. The `\jobname` can also be set on the command line when invoking (I)TeX. But either way, extra files are needed to support this workflow.

Also, in the context of a homework assignment, none of these methods allow the distribution of an answer template as a L^AT_EX source file. If the solutions are in the master TeX file, and conditionally typeset depending on options, they still remain the source.

4.3 Symlinks

In this workflow, a single TeX file is created, and each variant document is a symbolic link (or symlink) to the original file with a different file name. The operating system treats a symlink to a file as a different name for that file. Editing one of these “files” affects the contents referenced by the file and all of its symlinks. But the `\jobname` is determined by the file name, so it can be tested in order to conditionally execute certain code.

This avoids the colliding output file issue of command-line arguments, and is more lightweight than shell files. A new variant just requires a new symlink. It has the same disadvantages of the master-plus-shell files workflow, though. Code cannot be stripped out of the master file for a template, only gobbled and discarded. Not every operating system and file system has this capability, either.

4.4 Other preprocessors

DocStrip functions as a *preprocessor*—it converts one source file to another (or several others). There are other tools for this job, among them GPP and m4. Using another program requires installing, learning, and maintaining another program, whereas DocStrip is available wherever TeX is.

The DocStrip method described here is operating system independent. It is secure in that each document variant gets a distinct `\jobname` and output file name. It is lightweight in that only one DocStrip file needs to be created for every bundle of documents needed, and no programming other than TeX is necessary.

4.5 Disadvantages of the DocStrip method

One drawback of this method is that it requires an extra TeX run. First, the DocStrip file is compiled, extracting the various TeX files. Then each of them must be compiled. A bit of programming can automate this process (as well as decide when certain

files don't need to be recompiled), as we'll describe in the next section.

Not every \TeX editor can be used for DocStrip files. In particular, WYSIWYG or WYSIWYM editors such as LyX and TeXmacs expect the source file to be a regular (I^A) \TeX file. But any editor designed to operate on text files can be configured for DocStrip. Some of the most popular \TeX editors (for example, TeXShop, TeXworks, TeXstudio, AUCTEX) support DocStrip out of the box, as does Visual Studio Code.

Another, more uncomfortable drawback is that \TeX errors are only discovered when the generated \TeX files are compiled. The line numbers reported by \TeX at the time of the error are different from the line numbers in the DocStrip file. So diagnosing errors needs to be done without navigating to specific line numbers. Rather, the token list before the error can be used for a search string.

Finally, errors in the first run of \TeX (when DocStrip is extracting) can arise from unbalanced guard modifiers, e.g., a `%<*solutions>` line with no closing `%</solutions>`. These are hard to isolate since DocStrip does not log its processing with much context, and the error isn't discovered until the end of the file is reached.. I have been able to find these through a combination of retracing my steps, and selectively commenting out blocks.

5 Automation

To recap, this workflow requires editing a DocStrip file marked up as in Section 3, compiling that DocStrip file to generate separate \TeX files, then compiling the desired \TeX file. The first run can be done in any \TeX engine, because the batch declaration header (and `docstrip.tex`) is in core \TeX . The second set of compilations require whatever engine your destination documents require.

The `latexmk` program [2] works like *make* for \TeX projects. It examines a \TeX file to find dependencies, watches for warnings about re-running L^AT_EX, and runs the necessary commands to get the entire document stable. `latexmk` is written in Perl, distributed with \TeX Live and MiK \TeX , and actively maintained.

A one-line Unix command that does both of these is:

```
tex foo.dtx && latexmk
```

This processes `foo.dtx` and, upon success of that command, runs `latexmk`. Without file arguments, `latexmk` looks for any \TeX files in the current working directory and makes them. Any command-line options to `latexmk` (notably, `-pdf` to make sure the `pdflatex` engine is chosen, `-pdfxe` to ensure `xelatex`,

or `-pdflua` to ensure `lualatex`) will be passed when making each \TeX file.

The process can be further automated and integrated into various \TeX editors. I have written a `.latexmkrc` (configuration file for `latexmk`) that looks for DocStrip files in the argument list, and when found, parses their log files for names of generated files to make automatically. With this configuration, `latexmk foo.dtx` will take care of all generated files in one fell swoop.

Any editor that can run a program can be configured to run `latexmk`. For instance, I have written a TeXShop “engine” script that wraps around `latexmk` so configured. I edit the DocStrip and press Command-T once. I open one of the generated files in “preview” mode, which gives the PDF window but not the \TeX window (we won't be editing the generated file directly, so we don't need it). The preview window updates each time the underlying file is changed.

I have also gotten this workflow to succeed in Visual Studio Code with the L^AT_EX Workshop [3]. These script files and corresponding documentation are in the github repository.

6 Conclusion

I will continue to maintain and update the github repository referenced above. If you would like to try this method, and find that additional documentation would be useful, I will be happy to include it.

References

- [1] Caramdir. Passing parameters to a document, 2010. tex.stackexchange.com/q/1492
- [2] J. Collins, E. McLean, D. J. Musliner. `latexmk`—fully automated L^AT_EX document generation, 2019. ctan.org/pkg/latexmk
- [3] J. Lelong, T. Tamura, et al. Visual Studio Code L^AT_EX Workshop Extension, 2020. github.com/James-Yu/LaTeX-Workshop
- [4] meduz. What Makefile to produce slides and handouts [in] a common file?, 2014. tex.stackexchange.com/q/170542
- [5] F. Mittelbach, D. Duchier, et al. The DocStrip program, 2006. ctan.org/pkg/docstrip
- [6] reprogrammer. Passing command-line arguments to L^AT_EX document, 2009. stackoverflow.com/q/1465665
- [7] E. R. Scheinerman. *Mathematics: A Discrete Introduction*. Thomson Brooks/Cole, Belmont, MA, 2nd edition, 2005.

◊ Matthew Leingang
New York University
`leingang` (at) `nyu` dot `edu`
www.cims.nyu.edu/~leingang

L^AT_EX News

Issue 32, October 2020

Contents

Introduction	1	Changes to packages in the <code>tools</code> category	5
Providing <code>xparse</code> in the format	1	array: Support stretchable glue in w-columns	5
A hook management system for L^AT_EX	1	array: Use math mode for w and W-cells in array	6
Other changes to the L^AT_EX kernel	2	array: Fix for \firstline and \lastline	6
\symbol in math mode for large Unicode values	2	variorref: Support Japanese as a language option	6
Correct Unicode value of \=y (\bar{y})	2	xr: Support for spaces in filenames	6
Add support for Unicode soft hyphens	2	Changes to packages in the <code>amsmath</code> category	6
Fix capital accents in Unicode engines	2	Placement corrections for two accent commands	6
Support calc in various kernel commands	2	Fixes to aligned and gathered	6
Support ε -TeX length expressions in picture	2	Detect Unicode engines when setting	6
coordinates	2	\std@minus and \std@equal	6
Spaces in filenames of included files	2	lualatex-math: Use LuaTeX primitives	6
Avoid extra line in \centering, \raggedleft	3	Changes to the <code>babel</code> package	6
or \raggedright	3	Introduction	3
Set a non-zero \baselineskip in text scripts	3	The 2020-10-01 release of L ^A T _E X shows that work on	3
Spacing issues when using \linethickness	3	improving L ^A T _E X has again intensified. The two most	3
Better support for the legacy series default	3	important new features are the kernel support for <code>xparse</code>	3
interface	3	and the introduction of the new hook management	3
Support for uncommon font series defaults	3	system for L ^A T _E X, but as you can see there are many	3
Checking the current font series context	3	smaller enhancements and bug fixes added to the kernel	3
Avoid spurious package option warning	3	and various packages.	3
Adjusting fleqn	3	Providing <code>xparse</code> in the format	3
Provide \clap	3	The official interface in the L ^A T _E X 2 ε kernel for cre-	4
Fix to legacy math alphabet interface	4	ating document-level commands has always been	4
Added tests for format, package and class dates	4	\newcommand. This was a big step forward from	4
Avoid problematic spaces after \verb	4	L ^A T _E X 2.09. However, it was still very limited in the	4
Provide a way to copy robust commands	4	types of command it can create: those taking at most	4
... and a way to \show them	4	one optional argument in square brackets, then zero or	4
Merge l3docstrip into docstrip	4	more mandatory arguments. Richer syntaxes required	4
Support vertical typesetting with doc	4	use of the TeX \def primitive along with appropriate	4
Record the counter name stepped by	5	low-level macro programming.	5
\refstepcounter	5	The L ^A T _E X team started work on a comprehensive	5
Native LuaTeX behavior for \-	5	document-command parser, <code>xparse</code> , in the late 1990s. In	5
Allow \par commands inside \typeout	5	the past decade, the experimental ideas it provides have	5
Spacing commands moved from amsmath to	5	been carefully worked through and moved to a stable	5
the kernel	5	footing. As such, <code>xparse</code> is now used to define a very	5
Access raw glyphs in LuaTeX without	5	large number of document and package commands. It	5
reloading fonts	5	does this by providing a rich and self-consistent syntax	5
Added a fourth empty argument to	5	to describe a wide range of interfaces seen in L ^A T _E X	5
\contentsline	5	packages.	5
LuaTeX callback new_graf made exclusive .	5	The ideas developed in <code>xparse</code> are now sufficiently	5
Changes to packages in the <code>graphics</code> category	5	well tested that the majority can be transferred into the	5
Generate a warning if existing color definition	5	L ^A T _E X kernel. Thus the following commands have been	5
is changed	5	added	5
Specifying viewport in the graphics package	5	<ul style="list-style-type: none"> • \NewDocumentCommand, \RenewDocumentCommand, 	5
Normalizing \endlinechar	5	<ul style="list-style-type: none"> \ProvideDocumentCommand, 	5
Files with multiple parts	5	<ul style="list-style-type: none"> \DeclareDocumentCommand 	5

- `\NewExpandableDocumentCommand`,
`\RenewExpandableDocumentCommand`,
`\ProvideExpandableDocumentCommand`,
`\DeclareExpandableDocumentCommand`
- `\NewDocumentEnvironment`,
`\RenewDocumentEnvironment`,
`\ProvideDocumentEnvironment`,
`\DeclareDocumentEnvironment`
- `\BooleanTrue` `\BooleanFalse`
- `\IfBooleanTF`, `\IfBooleanT`, `\IfBooleanF`
- `\IfNoValueTF`, `\IfNoValueT`, `\IfNoValueF`
- `\IfValueTF`, `\IfValueT`, `\IfValueF`
- `\SplitArgument`, `\SplitList`, `\TrimSpaces`,
`\ProcessList`, `\ReverseBoolean`
- `\GetDocumentCommandArgSpec`
`\GetDocumentEnvironmentArgSpec`

Most, but not all, of the argument types defined by `xparse` are now supported at the kernel level. In particular, the types `g/G`, `l` and `u` are *not* provided by the kernel code; these are deprecated but still available by explicitly loading `xparse`. All other argument types are now available directly within the L^AT_EX 2_ε kernel.

A hook management system for L^AT_EX

With the fall 2020 release of L^AT_EX we provide a general hook management system for the kernel and for packages. This will allow packages to safely add code to various kernel and package hooks and if necessary define rules to reorder the code in the hooks to resolve typical package loading order issues. This hook system is written in the L3 programming layer and thus forms the first larger application within the kernel that makes use of the L^AT_EX3 functionality now available (if we discount `xparse` which has already been available for a long time as a separate package).

The file `lthooks.dtx` holds the core management code for hooks and defines basic hooks for environments (as previously offered by `etoolbox`), `ltshipout.dtx` provides kernel hooks into the shipout process (making packages like `atbegshi`, etc., unnecessary) and the file `ltfilehook.dtx` holds redefinitions for commands like `\input` or `\usepackage` so that they offer hooks in a similar fashion to what is provided by the `filehook` package.

At the moment the integration is lightweight, overwriting definitions made earlier during format generation (though this will change after more thorough testing). For that reason the documentation isn't in its final form either and you have to read through three different documents:

`lthooks-doc.pdf` Core management interface and basic hooks for environments provided by the kernel.

`ltshipout-doc.pdf` Hooks accessible while a page is being shipped out.

`ltfilehook-doc.pdf` Hooks used when reading a file.

For those who wish to also study the code, replace `-doc` with `-code`, e.g., `lthooks-code.pdf`. All documents should be accessible via `texdoc`, e.g.,

```
texdoc lthooks-doc
```

should open the core documentation for you.

Other changes to the L^AT_EX kernel

\symbol in math mode for large Unicode values

The L^AT_EX 2_ε kernel defines the command `\symbol`, which allows characters to be typeset by entering their ‘slot number’. With the LuaT_EX and X_ET_EX engines, these slot numbers can extend to very large values to accommodate Unicode characters in the upper Unicode planes (e.g., bold mathematical capital A is slot number “1D400 in hex or 119808 in decimal). The X_ET_EX engine did not allow `\symbol` in math mode for values above 2^{16} ; this limitation has now been lifted.

(github issue 124)

Correct Unicode value of \=y (ŷ)

The Unicode slot for \bar{y} was incorrectly pointing to the slot for \bar{Y} . This has been corrected. (github issue 326)

Add support for Unicode soft hyphens

For a long time, the UTF-8 option for `inputenc` made the Unicode soft hyphen character (U+00AD) an alias for the L^AT_EX soft hyphen `\-`. The Unicode engines X_ET_EX and LuaT_EX behaved differently though: They either ignored U+00AD or interpreted it as an unconditional hyphen. This inconsistency is fixed now and L^AT_EX always treats U+00AD as `\-`. (github issue 323)

Fix capital accents in Unicode engines

In Unicode engines the capital accents such as `\capitalcedilla`, etc., have been implemented as trivial shorthands for the normal accents (because other than Computer Modern virtually no fonts support them), but that failed when `hyperref` got loaded. This has been corrected. (github issue 332)

Support calc in various kernel commands

The `\hspace`, `\vspace`, `\addvspace`, `\`` and other commands simply passed their argument to a T_EX primitive to produce the necessary space. As a result it was impossible to specify anything other than a simple dimension value in such arguments. This has been changed, so that now `calc` syntax is also supported with these commands. (github issue 152)

Support ε-T_EX length expressions in picture coordinates

Picture mode coordinates specified with `(_,_)` previously accepted multiples of `\unitlength`. They now also allow ε-T_EX length expressions (as used by the `\glueexpr` primitive although all uses in picture mode are non-stretchy).

So, valid uses include `\put(2,2)` as previously, but now also uses such as

```
\put(\textwidth-5cm,0.4\textheight).
```

Note that you can only use expressions with lengths; `\put(1+2,0)` is not supported.

Spaces in filenames of included files

File names containing spaces lead to unexpected results when used in the commands `\include` and `\includeonly`. This has now been fixed and the argument to `\include` can contain a file name containing spaces. Leading or trailing spaces will be stripped off but spaces within the file name are kept. The argument to `\includeonly`, which is a comma-separated list of files to process, can also contain spaces with any leading and trailing spaces stripped from the individual filenames while spaces *in* the file names will remain intact.

(github issues 217 and 218)

Avoid extra line in \centering, \raggedleft or \raggedright

If we aren't justifying paragraphs then a very long word (longer than a line) could result in an unnecessary extra line in order to prevent a hyphen in the second-last line of the paragraph. This is now avoided by setting `\finalhyphendemerits` to zero in unjustified settings.

(github issue 274)

Set a non-zero \baselineskip in text scripts

As `\textsuperscript` and `\textsubscript` usually contain only a few characters on a single line the `\baselineskip` was set to zero. However, `hyperref` uses that value to determine the height of a link box which consequently came out far too small. This has been adjusted.

(github issue 249)

Spacing issues when using \linethickness

In some circumstances the use of `\linethickness` introduced a spurious space that shifted objects in a `picture` environment to the right. This has been corrected.

(github issue 274)

Better support for the legacy series default interface

In the initial implementation of L^AT_EX's font selection scheme (NFSS) changes to any default were carried out by redefining some commands, e.g., `\seriesdefault`. In 2019 we introduced various extensions and with it new methods of customizing certain parts of NFSS, e.g., the recommended way for changing the series default(s) is now through `\DeclareFontSeriesDefault` [1]. In this release we improved the support for legacy documents using the old method to cover additional edge cases.

(github issues 306,315)

Support for uncommon font series defaults

If a font family was set up with fairly unusual font series defaults, e.g.,

```
\renewcommand\ttdefault{lmvtt}
\DeclareFontSeriesDefault[tt]{md}{lm}
\DeclareFontSeriesDefault[tt]{bf}{bm}
```

then a switch between the main document families, e.g., `\ttfamily... \rmfamily` did not always correctly continue typesetting in medium or bold series if that involved adjusting the values used by `\mdseries` or `\bfseries`. This has now been corrected.

(github issue 291)

Checking the current font series context

Sometimes it is necessary to define commands that act differently when used in bold context (e.g., inside `\textbf`). Now that it is possible in L^AT_EX to specify different “bf” defaults based for each of the three meta families (`rm`, `sf` and `tt`) via `\DeclareFontSeriesDefault`, it is no longer easy to answer the question “am I typesetting in a bold context?”. To help with this problem a new command was provided:

```
\IfFontSeriesContextTF{\langle context\rangle}
  {\langle true code\rangle}{\langle false code\rangle}
```

The `\langle context\rangle` can be either `bf` (bold) or `md` (medium) and depending on whether or not the current font is recognized as being selected through `\bfseries` or `\mdseries` the `\langle true code\rangle` or `\langle false code\rangle` is executed. As an example

```
\usepackage{bm} % (bold math)
\newcommand\vbeta{\IfFontSeriesContextTF{bf}{%
  \ensuremath{\bm{\beta}}}{%
  \ensuremath{\beta}}}
```

This way you can write `\vbeta`-isotopes and if used in a heading it comes out in a bolder version.

(github issue 336)

Avoid spurious package option warning

When a package is loaded with a number of options, say X, Y and Z, and then later another loading attempt was made with a subset of the options or no options, it was possible to get an error message that option X is not known to the package. This obviously incorrect error was due to a timing issue where the list of available options got lost prematurely. This has now been fixed.

(github issue 22)

Adjusting fleqn

In `amsmath` the `\mathindent` parameter used with the `fleqn` design is a rubber length parameter allowing for setting it to a value such as `1em minus 1em`, i.e., so that the normal indentation can be reduced in case of very wide math displays. This is now also supported by the L^AT_EX standard classes.

In addition a compressible space between formula and equation number in the `equation` environment got added when the `fleqn` option is used so that a very wide formula doesn't bump into the equation number.

(github issue 252)

Provide \clap

L^AT_EX has inherited `\llap` and `\rlap` from plain T_EX (zero-sized boxes whose content sticks out to the left or right, respectively) but there isn't a corresponding `\clap` command that centers the material. This missing command was added by several packages, e.g., `mathtools`, and has now been added to the kernel.

Fix to legacy math alphabet interface

When using the L^AT_EX 2.09 legacy math alphabet interface, e.g., `\$\\sf -1$` instead of `\$\\mathsf{-1}$`, an extra math Ord atom was added to the formula in case the math alphabet was used for the first time. In some cases this math atom would change the spacing, e.g., change the unary minus sign into a binary minus in the above example. This has finally been fixed.

(gnats issue latex/3357)

Added tests for format, package and class dates

To implement compatibility code or to ensure that certain features are available it is helpful and often necessary to check the date of the format or that of a package or class and execute different code based on the result. For that, L^AT_EX previously had only internal commands (`\@ifpackagelater` and `\@ifclasslater`) for testing package or class names, but nothing reasonable for testing the format date. For the latter one had to resort to some obscure command `\@ifl@t@r` that, given its cryptic name, was clearly never intended for use even in package or class code. Furthermore, even the existing interface commands were defective as they are testing for “equal or later” and not for “later” as their names indicate.

We have therefore introduced three new CamelCase commands as the official interface for such tests

```
\IfFormatAtLeastTF{\(date\)}
  {\(true code\)}{\(false code\)}
```

and for package and class tests

```
\IfClassAtLeastTF{\(class name\)}{\(date\)}
  {\(true code\)}{\(false code\)}
\IfPackageAtLeastTF{\(package name\)}{\(date\)}
  {\(true code\)}{\(false code\)}
```

For compatibility reasons the legacy commands remain available, but we suggest to replace them over time and use the new interfaces in new code. (github issue 186)

Avoid problematic spaces after \verb

If a user typed `\verb!~!_foo` instead of `\verb!~! foo` by mistake, then surprisingly the result was “`!~!foo`” without any warning or error. What happened was that the `_` became the argument delimiter due to the rather complex processing done by `\verb` to render verbatim. This has been fixed and spaces directly following the command `\verb` or `\verb*` are now ignored as elsewhere.

(github issue 327)

Provide a way to copy robust commands...

With the previous L^AT_EX 2 _{ϵ} release, several user-level commands were made robust, so the need for a way to create copies of these commands (often to redefine them) increased, and the L^AT_EX 2 _{ϵ} kernel didn’t have a way to do so. Previously this functionality was provided in part by Heiko Oberdiek’s `letltxmacro` package, which allows a robust command `\foo` to be copied to `\bar` with `\LetLtxMacro{\bar}{\foo}`.

From this release onwards, the L^AT_EX 2 _{ϵ} kernel provides `\NewCommandCopy` (and `\Renew...` and `\Declare...` variants) which functions almost like `\LetLtxMacro`. To the end user, both should work the same way, and one shouldn’t need to worry about the definition of the command: `\NewCommandCopy` should do the hard work.

`\NewCommandCopy` knows about the different types of definitions from the L^AT_EX 2 _{ϵ} kernel, and also from other packages, such as `xparse`’s command declarations like `\NewDocumentCommand`, and `etoolbox`’s `\newrobustcmd`, and it can be extended to cover further packages.

(github issue 239)

... and a way to \show them

It is sometimes necessary to look up the definition of a command, and often one not only doesn’t know where that command is defined, but doesn’t know if it gets redefined by some package, so often enough looking at the source doesn’t help. The typical way around this problem is to use TeX’s `\show` primitive to look at the definition of a command, which works fine until the command being `\shown` is robust. With `\show\frac` one sees

```
> \frac=macro:
->\protect \frac .
```

which is not very helpful. To show the actual command the user needed to notice that the real definition of `\frac` is in the `\frac_` macro and do `\expandafter\show\csname frac\space\endcsname`.

But with the machinery for copying robust commands in place it is already possible to examine a command and detect (as far as a macro expansion language allows) how it was defined. `\ShowCommand` knows that and with `\ShowCommand\frac` the terminal will show

```
> \frac=robust macro:
->\protect \frac .
```

```
> \frac =\long macro:
#1#2->{\begingroup #1\endgroup \over #2}.
```

(github issue 373)

Merge l3docstrip into docstrip

The file `l3docstrip.tex` offered a small extension over the original `docstrip.tex` file supporting the `%<@@=(module)>`. This has been merged into `docstrip` so that it can now be used for both traditional `.dtx` files and those containing code written in the L³ programming layer language.

(github issue 337)

Support vertical typesetting with doc

The `macrocode` environment uses a `trivlist` internally and as part of this sets up the `\@labels` box to contain some horizontal skips, but that box is never used. As a result this generates an issue in some circumstances if the typesetting direction is vertical. This has now been corrected to support such use cases as well.

(github issue 344)

Record the counter name stepped by \refstepcounter
`\refstepcounter` now stores the name of the counter in `@currentcounter`. This allows packages like `zref` and `hyperref` to store the name without having to patch `\refstepcounter`.
(github issue 300)

Native LuaTeX behavior for \-

`LATEX` changes `\-` to add a discretionary hyphen even if `\hyphenchar` is set to `-1`. This change is not necessary under `LuaTeX` because there `\-` is not affected by `\hyphenchar` in the first place. Therefore this behavior has been changed to ensure that `LuaTeX`'s (language specific) hyphenation characters are respected by `\-`.

Allow \par commands inside \typeout

`\typeout` used to choke when seeing an empty line or a `\par` command in its argument. However, sometimes it is used to display arbitrary user input or code (wrapped, for example, in `\unexpanded`) which may contain explicit `\par` commands. This is now allowed.
(github issue 335)

Spacing commands moved from amsmath to the kernel

Originally `LATEX` only provided a small set of spacing commands for use in text and math; some of the commands like `\;` were only supported in math mode. `amsmath` normalized and provided all of them in text and math. This code has now been moved to the kernel so that it is generally available.
(github issue 303)

command name(s)	math	text
<code>\, \thinspace</code>	<code>xx</code>	<code>xx</code>
<code>\! \negthinspace</code>	<code>xx</code>	<code>xx</code>
<code>\: \> \medspace</code>	<code>xx</code>	<code>xx</code>
<code>\negmedspace</code>	<code>xx</code>	<code>xx</code>
<code>\: \thickspace</code>	<code>xx</code>	<code>xx</code>
<code>\negthickspace</code>	<code>xx</code>	<code>xx</code>

Access raw glyphs in LuaTeX without reloading fonts

`LATEX`'s definitions for `\textquotesingle`, `\textasciigrave`, and `\textquotedbl` for the TU encoding in `LuaTeX` need special handling to stop the shaper from replacing these characters with curly quotes. This used to be done by reloading the current font without the `tlig` feature, but that came with multiple disadvantages: It behaves differently than the corresponding `XeTeX` code and it is not very efficient. This code has now been replaced with an implementation which injects a protected glyph node which is not affected by font shaping.
(github issue 165)

Added a fourth empty argument to \contentsline

`LATEX`'s `\addcontentsline` writes a `\contentsline` command with three arguments to the `.toc` and similar files. `hyperref` redefines `\addcontentsline` to write a fourth argument. The change unifies the number of arguments by writing an additional empty brace group.
(github issue 370)

LuaTeX callback new_graf made exclusive

Corrected an incorrect callback type which caused return values from the `new_graf` callback to be ignored and paragraph indentation to be suppressed. In the new version, only one `new_graf` callback handler can be active at a time, which allows this handler to take full control of paragraph indentation.
(github issue 188)

Changes to packages in the graphics category

Generate a warning if existing color definition is changed
If a color is defined twice using `\DefineNamedColor`, no info text `Redefining color ... in named color model ...` was written to the log file, because of a typo in the check. This has been corrected.

(gnats issue graphics/3635)

Specifying viewport in the graphics package

Specifying a `BoundingBox` does not really have meaning when including non-EPS graphics in `pdfTeX` and `LuaTeX`. For some years the `graphicx` package `bb` key has been interpreted (with a warning) as a `viewport` key. This feature has been added to the two-argument form of `\includegraphics`, which is mostly used in the `graphics` package. `\includegraphics[1,2][3,4]{file}` will now be interpreted in `pdfTeX` and `LuaTeX` in the same way as `graphicx`'s
`\includegraphics[viewport=1 2 3 4]{file}`.

Normalizing \endlinechar

If `\endlinechar` is set to `-1` so that ends of lines are ignored in special contexts, then a low level `TeX` error would be generated by code parsing `BoundingBox` comments. The package now locally sets `\endlinechar` to its standard value while reading files.
(github issue 286)

Files with multiple parts

Sometimes one has a graphics file, say, `file.svg`, and converts it to another format to include it in `LATEX` and ends up with a file named `file.svg.png`. In previous releases, if the user did `\includegraphics{file.svg}`, an error would be raised and the graphics inclusion would fail due to the unknown `.svg` extension. The `graphics` package now checks if the given extension is known, and if it doesn't, it tries appending the known extensions until it finds a graphics file with a valid extension, otherwise it falls back to the file as requested.
(github issue 355)

Changes to packages in the tools category

array: Support stretchable glue in w-columns

If stretchable glue, e.g., `\dotfill`, is used in `tabular` columns made with the `array` package, it stretches as it would in normal paragraph text. The one exception was `w-columns` (but not `W-columns`) where it got forced to its nominal width (which in case of `\hfill` or `\dotfill` is `0 pt`). This has been corrected and now `w-columns` behave like all other column types in this respect.
(github issue 270)

array: Use math mode for w and W-cells in array

The `w` and `W`-columns are LR-columns very similar to `l`, `c` and `r`. It is therefore natural to expect their cell content to be typeset in math mode instead of text mode if they are used in an `array` environment. This has now been adjusted. Note that this is a breaking change in version v2.5! If you have used `w` or `W`-columns in older documents either add `>{$}...<{$}` for such columns or remove the `$` signs in the cells. Alternatively, you can roll back to the old version by loading `array` with

```
\usepackage{array}[=v2.4]
```

in such documents.

(github issue 297)

array: Fix for \firsthline and \lasthline

Replacing `\hline` with `\firsthline` or `\lasthline` could lead in some cases to an increase of the tabular width. This has now been corrected. (github issue 322)

varioref: Support Japanese as a language option

The package now recognizes `japanese` as a language option. The extra complication is that for grammatical reasons `\vref`, `\Vref`, `\vrefrange` and `\fullref` need a structure different from all other languages currently supported. To accommodate this, `\vrefformat`, `\Vrefformat`, `\vrefrangeformat`, and `\fullrefformat` have been added to all languages. (github issue 352)

xr: Support for spaces in filenames

The command `\externaldocument`, provided by `xr`, now also supports filenames with spaces, just like `\include` and `\includeonly`. (github issue 223)

Changes to packages in the amsmath category***Placement corrections for two accent commands***

The accent commands `\ddot` and `\dddot` (producing triple and quadruple dot accents) moved the base character vertically in certain situations if it was a single glyph, e.g., `$Q \ddot{Q}$` were not at the same baseline. This has been corrected. (github issue 126)

Fixes to aligned and gathered

The environments `aligned` and `gathered` have a trailing optional argument to specify the vertical position of the environment with respect to the rest of the line. Allowed values are `t`, `b` and `c` but the code only tested for `b` and `t` and assumed anything else must be `c`. As a result, a formula starting with a bracket group would get mangled without warning—the group being dropped and interpreted as a request for centering. After more than 25 years this has now been corrected. If such a group is found a warning is given and the data is processed as part of the formula. (github issue 5)

Detect Unicode engines when setting \std@minus and \std@equal

`amsmath` now detects the Unicode engines and uses their extended commands to define `\std@minus` and

`\std@equal`. This avoids a package like `unicode-math` having to patch the code in the begin document hook to change the commands.

lualatex-math: Use LuaTEX primitives

For a number of years `lualatex-math` patched `\frac`, `\genfrac` and the `subarray` environment to make use of new `luATEX` primitives. This code has now been integrated into `amsmath`.

Changes to the babel package

Multilingual typesetting has evolved greatly in recent years, and `babel`, like `LATEX` itself, has followed the footsteps of Unicode and the W3C consortia to produce proper output in many languages.

Furthermore, the traditional model to define and select languages (which can be called “vertical”), based on closed files, while still the preferred one in monolingual documents, is being extended with a new model (which can be called “horizontal”) based on *services* provided by `babel`, which allows defining and redefining locales with the help of simple `ini` files based on key/value pairs. The `babel` package provides about 250 of these files, which have been generated with the help of the Unicode Common Language Data Repository.

Thanks to the recent advances in `lualatex` and `luatofloat`, `babel` currently provides *services* for bidi typesetting, line breaking for Southeast Asian and CJK scripts, nonstandard hyphenation (like ff to ff-f), alphabetic and additive counters, automatic selection of fonts and languages based on the script, etc. This means `babel` can be used to typeset a wide variety of languages, such as Russian, Arabic, Hindi, Thai, Japanese, Bangla, Amharic, Greek, and many others.

In addition, since these `ini` files are easily parseable, they can serve as a source for other packages.

For further details take a look at the `babel` package documentation [4].

References

- [1] LATEX Project Team: *LATEX 2ε news 31*. <https://latex-project.org/news/latex2e-news/1tnews31.pdf>
- [2] *LATEX documentation on the LATEX Project Website*. <https://latex-project.org/help/documentation/>
- [3] *LATEX issue tracker*. <https://github.com/latex3/latex2e/issues/>
- [4] Javier Bezos and Johannes Braams. *Babel—Localization and internationalization*. <https://www.ctan.org/pkg/babel>

L^AT_EX Tagged PDF — A blueprint for a large project

Frank Mittelbach, Chris Rowley

Abstract

In Frank’s talk at the TUG 2020 online conference we announced the start of a multi-year project to enhance L^AT_EX to fully and naturally support the creation of structured document formats, in particular the “tagged PDF” format as required by accessibility standards such as PDF/UA.

In this short article we outline the background to this project and some of its history so far. We then describe the major features of the project and the tasks involved, of which more details can be found in the Feasibility Study [8] that was prepared as the first part of our co-operation with Adobe.

This leads on to a description of how we plan to use the study as the basis for our work on the project and some details of our planned working methodologies, illustrated by what we have achieved so far and leading to a discussion of some of the obstacles we foresee.

Finally there is also a summary of recent, current and upcoming activities on and around the project.

Contents

1	A short history of the project	292
2	What the project covers	293
3	Explaining the blueprint	294
3.1	The major tasks	295
3.2	The project plan	295
3.3	Timeline and other considerations .	296
4	First results	297
5	Outlook	297

1 A short history of the project

When T_EX was developed by Don Knuth in the early 1980s, it was designed as a high-quality automated typesetting system that concerned itself solely with the production of a “printed result” from text input, with paper as the final intended output medium. Any other kind of typeset output was either not supported or not directly supported.

Partially in parallel to Knuth’s work, Leslie Lamport developed L^AT_EX as a complex and highly integrated collection of T_EX macros that “runs on top of T_EX”. Some time later, during the 1990s, some support for graphics and color was added, but still following the same paradigm and aimed at printing

on physical media, now including overhead projector slides.

The move beyond paper came soon after this, with hyperlinks and other web publishing support getting layered on top of basic L^AT_EX; but these (together with many other subsequent extensions) were never incorporated as integral parts of its design. This piecemeal development has resulted in the extensions being very fragile so that they often break in more complex documents, especially when used in combination.

Even today it is the case that, in many areas, core L^AT_EX provides little in the way of APIs that are robust enough to be safely used or built on by developers. Therefore most such extensions are forced to contain a lot of code that overwrites many internal L^AT_EX commands: such unfortunate and unsustainable implementation practices being currently necessary in order to implement most extra functionality. Such methodology naturally leads to many issues, of which a common and particularly frustrating one is incompatibilities between extensions.

Another severe limitation of current L^AT_EX is that, whenever it produces an output page, it very carefully throws away the wealth of structural information it has used and accumulated while producing that page.¹ As a result, a PDF or DVI file produced by basic L^AT_EX is nothing but a stream of positioned glyphs containing very little structural information.

As long as your intention is only to print a document on a physical medium, then this is all that is required. However, for quite a while now other uses of documents have been increasing in importance so that nowadays many documents are never printed, or printed only as a secondary consideration.

Coming into the 21st century, for many reasons great interest has arisen in the production of PDF documents that are “accessible”, in the sense that they contain information to assist screen reading software, etc., and, more formally, that they adhere to the PDF/UA (Universal Accessibility) standard [17], explained further in [3]. Ross Moore has recently written a comprehensive introduction to this subject [9].

¹ This jettisoning of much useful information was absolutely essential during the early decades of L^AT_EX development because the system would otherwise never have run in the severely limited memory available on the then current computer systems. So all information no longer needed for producing a “printed page” was dropped immediately so as to keep the memory footprint within reasonable limits.

But even that was not always enough: on his first PC one author was greeted with “out of memory” while loading the `article` class.

One important requirement for such PDF documents is that they must contain a significant amount of embedded structural information and other data. At the moment, all methods of producing such “accessible PDFs”, including the use of L^AT_EX, require extensive manual labor in preparing the source or in post-processing the PDF (maybe even at both stages); and these labors often have to be repeated after making even minimal changes to the (L^AT_EX or other) source. Such methods and their inherent problems have been discussed and demonstrated recently by developers such as Ross Moore, much of whose work in this area can be accessed through the web site at [10].

The L^AT_EX Project Team have for some years been well aware that these new usages are not adequately supported by the current system architecture of L^AT_EX, and that major work in this area is therefore urgently needed to ensure that L^AT_EX remains an important and relevant document source format. However, the amount of work required to make such major changes to the L^AT_EX system architecture will be enormous, and definitely way beyond the regular resources of our small team of volunteers working in their spare time! Or maybe just about possible, but only given a very long—and most likely too long—period of time.

At the TUG conference 2019 in Palo Alto our previously pessimistic outlook on this subject became cautiously optimistic, because we were approached by two senior engineers from Adobe to discuss the possibility of producing structured PDF from L^AT_EX source without the need for the normal requirement of considerable manual post-processing. Moreover, they had come to ask us what it would take to make this happen. They indicated that, if this turned out to be a feasible project for which we could provide them with a convincing project plan, then they would happily advise Adobe, at a suitably influential level of management, to support such work, both financially and in other ways—no strings attached.²

² Adobe's rationale for this support is that L^AT_EX is currently a very important document format in the STEM disciplines and is one of the formats best suited to produce well-structured PDF “out of the box”, given that its source already contains such a large amount of information about the document's structure. Thus, with L^AT_EX capable of supporting the straightforward production of structured PDF documents, a large number of such documents would become available on the Internet. In addition, many older documents could be reasonably easily reprocessed to add this structural information to their PDF form. For Adobe all this is of great interest as the existence of large numbers of such documents forms an important and strategic input to their Document Cloud [1] offerings and related future services.

As a result of these discussions, towards the end of 2019 the authors, together with Ulrike Fischer, produced an extended feasibility study for the project, aimed primarily at Adobe engineers and decision makers. This study [8] describes in some detail the various tasks that will constitute the project and their inter-dependencies. It also contains a plan covering how, and in what order, these tasks should be tackled in order both to achieve the final goal and, at the same time, to provide intermediate concrete results that are relevant to user communities (both L^AT_EX and PDF); these latter will help in obtaining feedback that is essential to the successful completion of later tasks.

The project plan gained the approval of the Adobe management and we agreed in early 2020 to start the project with Adobe contributing a substantial portion of the expected project costs. But this was 2020, so then “Corona arrived” with its huge spanner to throw into the works! Thus all activity came to a halt while Adobe was forced to reassess such external financial commitments. However, we were quite quickly told that this was a temporary setback, so we restarted work on the project, albeit at a slightly slower pace, and so were able to present our first results at the TUG 2020 online conference: i.e., the new hook management system for L^AT_EX.³

2 What the project covers

This project has the title “L^AT_EX Tagged PDF” so before discussing further its coverage we should describe the use of the word “Tagged” here. A good starting point for this is the following quote from the PDF standard [2, 15] which defines tagging in the PDF context thus:

Tagged PDF ([introduced in]PDF 1.4) is a stylized use of PDF that builds on the logical structure framework described in Section [...], “Logical Structure.” It defines a set of standard structure types and attributes that allow page content (text, graphics, and images) to be extracted and reused for other purposes. It is intended for use by tools that perform the following types of operations:

- Simple extraction of text and graphics for pasting into other applications
- Automatic reflow of text and associated graphics to fit a page of a different size than was assumed for the original layout
- Processing text for such purposes as searching, indexing, and spell-checking

³ The project task for this is described in section 2.2.5 of the Feasibility Study [8].

- Conversion to other common file formats (such as HTML, XML, and RTF) with document structure and basic styling information preserved
- Making content accessible to users with visual impairments (see Section [...], “Accessibility Support”)

Based on this definition we can now give a high-level description of this project by analyzing the quoted text from the perspective of how L^AT_EX currently works and how we will need to change it. The quoted text tells us that to produce a Tagged PDF document from a L^AT_EX source we must proceed as follows:

First, we must exploit L^AT_EX’s knowledge of how the components of a document are presented in the L^AT_EX source file and used to describe its structure, consisting of “structure types and attributes” such as the paragraph text, headings, lists, items, graphics, tables, table cells, mathematics, etc.

Then, we must use this knowledge about the source document to add the necessary components to the PDF we are constructing. There are two basic ingredients of this: a tree structure whose nodes represent the “logical components” of the document; and a mechanism for identifying those portions of the “page content” that correspond to each (leaf) node in this tree.

These “tagged” portions of the content, together with information about the corresponding structure node, can then be used by other (consumer) applications to carry out operations like those listed at the end of the quote above.

None of the above may sound too complicated given that L^AT_EX knows what kind of structures will be in the source file that it is processing. But as we explained earlier, for performance reasons (necessary when L^AT_EX was designed and implemented), currently this structural information is disposed of as soon as possible—therefore, by the time of writing out the content of a page to the PDF file it is no longer available.

In addition, there are some problematic details within these processes. For example, the main content of a PDF document is a collection of “page objects” and the “tagged portions” of the text have to be nested within each page object; but T_EX’s asynchronous pagination process does not play well with this constraint. Dealing with this problem is therefore part of one of the lower-level tasks in the project, one that may best be solved by enhancements at the engine level.

For this reason, a large portion of the necessary project work consists of adding to the core of L^AT_EX all the functionality, data structures and interfaces that are necessary for preserving and processing the many details of this structural information. This will result in a much enhanced L^AT_EX system that can manage and transport structural information from a source to an output format.

With all of this firmly embedded into L^AT_EX, the project will be able to go further to provide, for example, appropriate document-level interfaces for some types of data that currently are not, at least by default, made explicit in the source file. In this area there already exist some solutions for aspects such as metadata, PDF bookmarks, etc., produced by Ross Moore, Scott Pakin, Heiko Oberdiek and others [11, 12, 14]. Wherever feasible, we shall unify and incorporate such useful existing work. Support for some, but not all, of the wider (beyond tagging) requirements of “Accessible PDF” will also be included in the project work.

We also want to include here some important (to us, at least) observations concerning the processing of structural information using L^AT_EX. From the document-level (frontend) processing to the internal data structures and interfaces, everything we wrote in the high-level description above is essentially independent of the target output format (Tagged PDF in this project). Identical processing will be needed for any output format that needs such structural information, including HTML, XML and other “tagged/structured formats”.

Thus, while the project will focus primarily on PDF output (either generated directly by the T_EX engine or through a DVI-based workflow) it will, as a bonus, make it easy to add other such output formats to the workflow by simply replacing an output (backend) module. Instead of PDF output, HTML5 or some other format will then get written. As of now, such alternative backends are not part of the project coverage, but once L^AT_EX is able to pass structure information with well-defined interfaces to a backend we expect that support for other structured output formats will follow. Such work may be undertaken by us or by other teams, possibly in parallel to the later phases of the project discussed in this article.

3 Explaining the blueprint

The document entitled “L^AT_EX Tagged PDF Feasibility Evaluation” [8], available from the L^AT_EX Project website [5], is a forty-page study that explains in detail both the project goals and the tasks that need to be undertaken, concluding with a plan for how we think the project should be undertaken. Thus it is

divided into three parts, beginning with an “Introduction” which contains an overview of the benefits of the project and goes on to explain why L^AT_EX documents make a good starting point for the production of tagged PDF.

It is advisable, if you are consulting this study, to bear in mind that it was addressed primarily to an audience within Adobe;⁴ this consisted of engineers and managers with a wide knowledge of digital typography and electronic publishing but not necessarily much background within the specialized world of T_EX, L^AT_EX and friends.

3.1 The major tasks

Following the overview, the next part of the study (“Project Overview”) documents all the major tasks that we consider necessary to reach the project goals. These tasks are categorized under five headings, the two most important being “General L^AT_EX Extension Tasks” and “Structured PDF Tasks”.

For each task, the study describes the rationale for its inclusion and the work that has to be undertaken. For the larger tasks, suitable subtasks are also identified. Furthermore, to support the project timeline planning, all the dependencies of each task on other tasks are documented. Each such task section ends with a list of deliverables, which will help us to measure, and hence monitor, our progress through the project.

Note that in total there are twenty engineering tasks for which we believe that the necessary work is well understood and therefore it “only” has to be done — BUT done very well! In addition there are a few project tasks that will require more extensive research. Their descriptions in the study are much less complete; this is simply because the engineering details are not as yet known, or not well enough understood. These research tasks are, naturally, likely to identify further engineering tasks.

Note that, since the tasks in this part are arranged by category, they are not listed in their chronological order of execution within the overall plan: i.e., a task’s number is no indication of how far along in the project that task will be tackled.

3.2 The project plan

In the final major part of the study (“Project Timeline”) we develop a complete project plan consisting of six phases. The tasks have been sorted into these phases in such way that the dependencies between them are respected; but also, and more importantly,

⁴ The public, cited, version of the study was updated in September 2020 with some minor redactions, corrections and clarifications.

we have ensured that the results of each phase will offer immediate benefits to the L^AT_EX community.

This latter requirement is important since it will lead to timely feedback and early adoption. As you can see below, we expect, for example, that on completion of phase II it will already be possible to automatically generate tagged PDFs for a restricted set of documents. In later phases this level of automation will be extended to a wider range of documents.

Phase I—Prepare the ground

This phase covers some tasks that are important precursors of later work; it is already well under way. One important deliverable here is the new general hook management system for L^AT_EX that was presented at the 2020 TUG conference and which is now part of core L^AT_EX, starting from the October 2020 release.

Another task here is the initial work on the low-level requirements for the creation of tagged PDF; this is currently available in the highly experimental package `tagpdf`, which uses prototype code from `pdfresources`-related files for the creation and management of PDF objects. This package thus supports such tasks as the creation in the PDF of the structure tree and adding the necessary connections between the nodes in this tree and the “content stream” of each page.

It currently enables the creation (with some user intervention) of tagged PDF documents; *but*, please note that this work is truly experimental and so both the code and interfaces are likely to change, or even vanish, at any time.

This phase is the only one that was not designed to offer significant benefits to users/authors. Nevertheless, the standardization and interfaces provided by the general hook management will arguably benefit these groups in addition to its direct target, the package developers.

Phase II—Provide tagging of simple documents

The main goal of phase II is to provide basic support for the automation of the tagging process. In this phase the automation will cover only relatively simple documents: those that do not contain any of the more complicated structures such as mathematics and tables.

At this stage, the automated tagging will be provided by loading an add-on package rather than its being in the kernel, and it will be only a prototype implementation.

For this phase several “workarounds” will be required in order to provide necessary, but missing,

features. For example, instead of setting up (as part of this phase) a full extended cross-reference system (similar to the `zref` package by Heiko Oberdiek, but built into core L^AT_EX) we will either use that package or add temporary code written to “work around” any issues resulting from this functionality not yet being a component of the kernel. Any such “workaround code” will need to be removed or adjusted in a later phase.

This ordering of the work was chosen so as to offer tangible results reasonably early in the life of the project, rather than taking up too much time at this stage on making purely internal improvements that have no immediately visible application and therefore appear to many users to be devoid of value.

Phase III—Remove the workarounds needed for tagging

The main goal of phase III is to extend the coverage of automatic tagging to a wider variety of documents by making further basic document elements “tagging aware”. These extensions will also enable us to remove the “workarounds” introduced (to provide a working prototype) in the previous phase. This phase will also include a metadata mechanism that is integrated into the L^AT_EX kernel.

Phase IV—Make basic tagging and hyperlinking available

The main goal of phase IV is to incorporate into the kernel all the code from the prototype packages. This needs to be done very carefully and cautiously as there should be no negative impact on the processing of legacy documents. Thus we expect to need at least one full L^AT_EX release cycle⁵ to complete this work.

Phase V—Extend the tagging capabilities

With basic tagging available, the focus of phase V is to provide extended support for tagging by adding tables and formulas (mathematics) to the supported document structures.

At this stage, interfaces will also be added wherever needed to support other aspects of Tagged PDF such as the specification of alternate text (for formulas, illustrations, etc.).

Phase VI—Handle standards

Finally, phase VI focuses on the provision of support for the relevant PDF standards (such as PDF/A [16], explained further in [13]; and PDF/UA [17], see also [3]) at least so far as this is possible using L^AT_EX

directly, i.e., without any post-processing of the PDF file. It also covers kernel support for some further features such as the production of outlines for a document and the incorporation into a PDF of “associated files”.

Research work

In addition to these six phases, which contain only tasks that are largely understood from a technical perspective, there are a number of other tasks that will require preparatory research in order to understand what engineering work is needed. This research will be carried out in parallel with the other work and, as indicated earlier, it is very likely to lead to the specification of additional engineering tasks. Thus the research outcomes will probably lead to some alterations and extensions in phases IV to VI.

Alterations, adjustments and reporting

On the whole we believe that this plan offers a consistent and sensible approach to attaining the intended goals of the project. However, there will undoubtedly be some changes since, for example, the research tasks will probably lead to some augmentation of the planned tasks and changes to some of them. Moreover, we have already found that we want to move some deliverables (from the larger tasks) forward to earlier phases. Also, in some cases we are thinking about not completely finishing a task within the planned phase because it appears more important to first focus on other work (of course, such changes will work only in cases where those deliverables are not required for other tasks). Furthermore, working through a task may identify some additional work that is needed but not yet accounted for, leading to new subtasks with extra deliverables.

We are therefore planning to provide an appropriate level of continuing (public) reports on the project’s status and results, including any additions or changes of direction. The Feasibility Study, with its fairly detailed documentation and deliverables, will provide a good starting point for this important activity. The exact form and shape of this reporting is not yet decided, but we will in due time announce it on the L^AT_EX Team website [5] and/or in *TUGboat* as appropriate.

3.3 Timeline and other considerations

The time estimates we supplied (in the Feasibility Study) for the individual tasks are based on the assumption that there will be *at least* (the equivalent of) one software developer working exclusively and full time on a task throughout its allocated time period, together with additional support from further

⁵ See Section 3.3 for further information concerning phases and this release cycle.

developers as necessary. We are also very well aware that activities such as documentation and testing require substantial specialized professional input that will also need to be resourced.

Whether or not this level of effort will be possible at all times will depend on various factors, an important one being the availability of appropriate funding that can free the responsible developers and other key workers from the need to undertake other work to earn a living. While the financial sponsorship from Adobe will go a long way towards meeting the needs of the project, it may well not be sufficient by itself to fully sustain the work. For all the above reasons, the estimates in the study should be considered as approximations and not as definite values.

A realistic scenario would be that each phase takes between one and two release cycles of L^AT_EX, of which there are two per year. This implies that the project will stretch across four years as a minimum, but it most probably will be somewhat longer. Additional funding will help to ensure timely delivery of each phase and may additionally allow us to broaden the scope in some areas. Nevertheless, given the complexity of the topic, any expectation of earlier delivery dates is not realistic.

It is also important to note that all the necessary updates to important external packages (those not supported by the L^AT_EX Project Team) are expected to be done using external resources, i.e., by the maintainers of those packages. This assumption is probably not tenable in all cases (see, for example, the discussion in [8, Task 2.4.3]). In such cases, additional work will have to be undertaken as part of the project, which will also alter the timeline or require hiring additional developers to take on such work.

Last but not least, the success of the project will also depend very much on productive collaborations with many people in the L^AT_EX community: testers, package writers, and possibly also those who tend to the underlying T_EX engines and the various utility programs used to produce PDF output. Also, from the wider world, we shall need input from a variety of experts in the production of high-quality PDF and from those with knowledge of how consumer applications use its features in the real world.

4 First results

As mentioned earlier, we have started the project despite COVID-19 getting in the way. So we can now already report on some success stories:

- The new L^AT_EX hook management system (Task 2.2.5 of [8]) presented at TUG 2020 [7] and in a *TUGboat* article [6];

- PDF string support (Task 2.2.1 of [8]) which is largely an internal enabler for later tasks;
- Initial experiments with `tagpdf` prototype code (part of Task 2.2.6 of [8]). See [4] for a discussion.

5 Outlook

Right now we are in the middle of phase I and expecting it to be finished with the Spring release of L^AT_EX in 2021. Given that we intend to shift the L^AT_EX release dates next year to better align with the yearly T_EX Live distributions, we expect to start work on some tasks of phase II already before the next L^AT_EX release.

In addition to the ongoing work on engineering tasks, some effort will go into managerial tasks, e.g.,

- Setting up connections and collaboration with external experts to gain expertise in areas where the present team is lacking, and to ensure that our work will continue to be focused on pertinent goals;
- Looking for additional financial support to bring in extra expertise and hence, we hope, speed up the later phases;
- Setting up a professional system for the production of documentation, high in both quality and quantity.

Of course, L^AT_EX development and maintenance will not be solely restricted to this project over the coming years. There are many other activities that the L^AT_EX Project Team plans to carry out in parallel. Their results will, as in the past, appear via the usual communication channels, e.g., our website, `ltnews` newsletters, *TUGboat* articles and/or Internet-based discussions on StackExchange chat, L^AT_EX-L and other places.

References

- [1] Adobe Document Cloud. https://en.wikipedia.org/wiki/Adobe_Document_Cloud.
- [2] Adobe Systems Inc. *PDF Reference 1.7*, Nov. 2006. https://www.adobe.com/devnet/pdf/pdf_reference.html. Freely available version of [15].
- [3] O. Drümmer, B. Chang. *PDF/UA in a Nutshell — Accessible documents with PDF*. PDF Association, Aug. 2013. <https://pdfa.org/resource/pdfua-in-a-nutshell/>.
- [4] U. Fischer. Creating accessible pdfs with L^AT_EX. *TUGboat* 41(1):26–28, 2020. <https://tug.org/TUGboat/tb41-1/tb127fischer-accessible.pdf>

- [5] L^AT_EX Project Team. Website of the L^AT_EX Project. <https://latex-project.org/>.
- [6] F. Mittelbach. Quo vadis L^AT_EX(3) Team—A look back and at the upcoming years. *TUGboat* 41(2):201–207, 2020. <https://latex-project.org/publications/indexbyyear/2020/>.
- [7] F. Mittelbach. Video: Quo vadis L^AT_EX(3) Team—A look back and at the upcoming years. 2020. Recording of the talk given at the online TUG 2020 conference, <https://youtu.be/zNci4lcB8Vo>.
- [8] F. Mittelbach, U. Fischer, C. Rowley. *L^AT_EX Tagged PDF Feasibility Evaluation*. L^AT_EX Project, Sept. 2020. <https://latex-project.org/publications/indexbyyear/2020/>.
- [9] R. Moore. Implementing PDF standards for mathematical publishing. *TUGboat* 39(2):131–135, 2018. <https://tug.org/TUGboat/tb39-2/tb122moore-pdf.pdf>
- [10] R. Moore. Website: Tagged PDF examples and resources, 2020. <http://maths.mq.edu.au/~ross/TaggedPDF> and in particular /TUG2019-movies.
- [11] R. Moore, C. V. Radhakrishnan, et al. Generation of PDF/X- and PDF/A-compliant PDFs with pdfTEX—pdfx.sty. <https://ctan.org/pkg/pdfx>, Mar. 2019.
- [12] H. Oberdiek. The bookmark package. <https://ctan.org/pkg/bookmark>, Dec. 2019.
- [13] A. Oettler. *PDF/A in a Nutshell—PDF for long-term archiving*. PDF Association, May 2013. <https://pdfa.org/resource/pdfa-in-a-nutshell/>.
- [14] S. Pakin. The hyperxmp package. <https://ctan.org/pkg/hyperxmp>, Oct. 2020.
- [15] Technical Committee ISO/TC 171/SC 2. *ISO 32000-1:2008 Document management—Portable document format (PDF 1.7)*, July 2008. <https://iso.org/standard/51502.html>. Freely available as [2].
- [16] Technical Committee ISO/TC 171/SC 2. *ISO 19005-2:2011 Document management—Document file format for long-term preservation—Part 2: Use of ISO 32000-1 (PDF/A-2)*, July 2011. <https://iso.org/standard/50655.html>.
- [17] Technical Committee ISO/TC 171/SC 2. *ISO 14289-1:2014 Document management applications—Electronic document file format enhancement for accessibility—1: Use of ISO 32000-1 (PDF/UA-1)*, Dec. 2014. <https://iso.org/standard/64599.html>.
- ◊ Frank Mittelbach
Mainz, Germany
`frank.mittelbach (at)
 latex-project dot org`
<https://www.latex-project.org>
- ◊ Chris Rowley
Sattahip, Chonburi, Thailand
`chris.rowley (at)
 latex-project dot org`
<https://www.latex-project.org>

Functions and expl3

Enrico Gregorio

Abstract

In this tutorial we discuss `expl3` functions, their role, definition and variants, also touching on variables.

1 Introduction

The term *function* is not used in standard L^AT_EX, but is a very important concept in the `expl3` language. The language itself had no precise name until a couple of years ago, when it was eventually decided that its name would be `expl3` just like the package that provided it (now merged in the L^AT_EX kernel).

In the language, care is taken to distinguish between *variables* and *functions*. Variables store some value that can change during the L^AT_EX run, whereas functions perform some action.

A special kind of variable is the *constant*, whose value is supposed not to change during the run. Well, the name seems to conflict with the nature, but mathematicians are used to this kind of stretched terminology: you who are not a mathematician, don't worry and carry on, just smile at mathematicians' bizarre way of thinking.

We'll be mostly interested in functions, but variables can be the staple food of functions, so we'll also need to know a bit about them.

Let me give an example using legacy concepts. The standard document classes, and most nonstandard ones as well, have the *command* `\title`. How does this work? This very paper has

```
\title{Functions and \explii}
```

at its start. When L^AT_EX processes this instruction, it will do something like

```
\gdef\@title{Functions and \explii}
```

so it will be able to use `\@title` when doing its typesetting job related to `\maketitle`.

There is a big conceptual difference between `\title` and `\@title`. The former performs an action, the latter is simply a container. In `expl3` terms, the former will be a function, the latter a variable: this function's action is to store a value in the specified variable.

At the user level the distinction is blurred; with

```
\newcommand{\CC}{\mathbb{C}}
```

is `\CC` be a function or a variable? Fortunately, it's not important to decide, because this is essentially a user's *shorthand* and at this level the distinction is almost irrelevant.

Simultaneously published in Italian for the GuIT 2020 conference, guitex.org.

The issue comes up when *programming*. In 'correct' L^AT_EX programming we have a user level command which calls a function that performs an action:

```
\NewDocumentCommand{\title}{m}
{
  \example_title:n { #1 }
}
%
\tl_new:N \g_example_title_tl
%
\cs_new_protected:Nn \example_title:n
{
  \tl_gset:Nn \g_example_title_tl { #1 }
}
```

Then the user level command `\maketitle` will use the value stored in the variable, via other functions.

This (imaginary) example shows many of the concepts we'll be discussing. We define a user-level command in terms of a function; this function has one argument (the given title) and its job is to set the value of a particular variable to the specified value. The variable has been declared in advance:

- `\title` is a user-level command; these are not the subject of this paper;
- `\g_example_title_tl` is a variable, defined using `\tl_new` (*tl* = token list);
- `\example_title:n` is a function, defined using `\cs_new_protected` (*cs* = control sequence).

We'll be discussing all of this in detail.

2 Naming conventions

A common problem with T_EX is that it has no concept of *namespace*, which only became common in computer science circles much later. Name conflicts were frequent in the olden days of L^AT_EX, and such conflicts still appear now and then. It might be appealing for a package to use `\@x` and `\@y` for coordinates, but package authors should be aware that if a simple name appeals to them, other authors have probably thought the same.

In `expl3`, variables should have a name of the form

```
\l_<prefix>_<proper name>_<type>
\g_<prefix>_<proper name>_<type>
\c_<prefix>_<proper name>_<type>
```

where the distinct parts are *important* and *necessary*:

- `l`, `g` and `c` declare that the variable is local, global or constant, respectively;
- `<prefix>` should be a unique string of letters for the package we're writing or the code in the document;
- `<proper name>` is an arbitrary string of letters possibly split into parts separated by an underscore;

- $\langle type \rangle$ is the type of variable.

The most common types of variables are:

- `t1`, for *token list*;
- `seq`, for *sequence*;
- `clist`, for *comma list*;
- `prop`, for *property list*;
- `int`, for *integer*;
- `dim`, for *dimension*;
- `box`, for *box*;
- `fp`, for *floating point*.

There are several others, but as the purpose of this tutorial is to talk about functions, I'll skip the more esoteric ones for now, only touching them when need comes.

Function names are similar:

```
\langle prefix\rangle_\langle proper name\rangle:\langle signature\rangle
```

The $\langle prefix \rangle$ and $\langle proper name \rangle$ are the same as before, but the $\langle signature \rangle$ must be explained. It can be an arbitrary string of characters among

```
c e f n N o v V w x T F
```

Each character given, except `w`, denotes an argument to the function. We'll be going into details soon.

Mathematical functions can depend on one or more arguments (well, also zero, but then they're constant functions) and the same is true for `expl3` functions. The purpose of the signature is to precisely specify how many arguments the function depends on and their type. For instance, the commonly used function

```
\seq_set_split:Nnn
```

takes three arguments, one of type `N` and two of type `n` *in that order*. An argument of type `N` should be a single token, the nature of which depends on the function; in the above case, it should be a sequence variable. An argument of type `n` should be a braced list of tokens. In our example, the title of the paper is an `n`-type argument to `\title`. This is a bit stretched, but should explain the concept.

The `w` type is an exception, because it specifies nothing except that the arguments to the function are *weird*, and one must refer to the package/code documentation in order to know how many there are and what syntax they have. Generally speaking, `w`-type arguments should only appear in low-level functions.

A call to the previously mentioned function might be something like

```
\seq_set_split:Nnn
  \l_example_test_seq
  { || }
  { a || b || c }
```

(more likely on one line in a source file; split here because of the paper's formatting). It doesn't matter now to know what this code does; `seq_set_split` is usually called as part of other processing and receives the arguments from other calls. The important thing is to see that the arguments follow the naming convention: the first one is a single token, the other two are braced lists of tokens.

A function can have no arguments, but the colon is still required. Although `TeX` will not balk if you define a function with a nonconforming name, sticking to the convention will help to avoid conflicts and to have more easily parseable code. A typical function with no arguments is `\scan_stop:`, which is nothing other than our old friend `\relax`. Maybe the `expl3` name is less poetic, but it expresses what the function's main purpose is.

3 Defining functions

There are many kernel functions which define functions. All of them share the `cs` prefix. The main ones are

```
\cs_new:Nn
\cs_new_protected:Nn
```

We'll discuss the others later. According to the naming conventions given, we can see that both have two arguments: the first argument is a single token, the name of the function to be defined, the second being the *replacement text*, that is, the code that will be substituted to the function's call.

While `expl3` tries hard to emulate a functional language, it is still implemented in `TeX`, which only knows primitives, macros and registers. This fact needs to be kept in mind when programming. On the other hand, the new language makes for simpler constructions, avoiding the clumsy (or fun, depending on the programmer's attitude) chains of `\expandafter` or `\noexpand` often seen in traditional `TeX`, understanding which is often quite hard.

A simple example is the internal function for managing the document's title, which we saw earlier:

```
\cs_new_protected:Nn \example_title:n
{
  \t1_gset:Nn \g_example_title_t1 { #1 }
}
```

Since the function's name has signature `:n` (strictly speaking, the colon is not part of the signature, but it's convenient to use it as a marker), `expl3` knows that the replacement text can use `#1` to refer to the argument supplied at call time.

Why are we using the `protected` instruction and not the simpler `cs_new`? Because our function will *set* the value of a variable. This is something

that for years has frustrated a horde of L^AT_EX programmers and has required the distinction between robust and fragile commands. Nowadays the issue is less relevant because almost all fragile commands have been ‘robustified’, but it can still bite.

What is the problem? If we define, in legacy L^AT_EX, something like

```
\newcommand{\foo}{1}%
  \renewcommand{\baz}{#1}%
}
```

which is the common way to store a value into a macro, and then somehow \foo ends up in \write or \edef, even under their wrappers \protected@write or \protected@edef, a long list of error messages appears. The traditional way of avoiding this is to use \DeclareRobustCommand instead of \newcommand.

Any function that works by setting variables or calling other protected functions should generally be protected itself. In case of doubt, protect.

Beware! The signature of the function to be defined can only consist of the characters N or n. Well, T and F are also allowed, but these are a special topic that we’ll touch later on. How do the other characters listed above get into signatures? This is a good question!

3.1 Generating variants

Suppose we’re doing a general purpose function for setting t_l (token list) variables to contain some material we need to use at later points. The user interface would utilize \setvar for storing the value and \usevar for delivering the value.

We face a problem: how can the user specify the name of a variable inside the document where the expl3 names are not allowed? Indeed, in normal text, the underscore cannot be used in a command name, so something like

```
\setvar{\l_example_var_a_t1}{something}
```

would bomb out. We’d like instead that the user types in

```
\setvar{a}{something}
\usevar{a}
```

(at different points of the document, of course). Let’s proceed at a slow pace. We’ll define the user interface afterwards. First we define a function that allocates a variable and stores a value in it; then a function that delivers the contents of a variable:

```
\cs_new_protected:Nn \example_setvar:Nn
{
  \tl_clear_new:N #1
  \tl_set:Nn #1 { #2 }
}
\cs_new:Nn \example_usevar:N
```

```
{
  \tl_use:N #1
}
```

The `tl_clear_new` instruction clears a possible preceding value or allocates a new variable. Then the `tl_set` function does the setting job; it’s not protected because it does no dangerous processing. However, this does not solve the problem we face. Here’s where the concept of *variants* comes into the scene. We do

```
\cs_generate_variant:Nn \example_setvar:Nn
  { cn }
\cs_generate_variant:Nn \example_usevar:Nn
  { c }
```

which effectively defines two *new* functions named

```
\example_setvar:cn
\example_usevar:c
```

What does c mean? It means that the new functions expect a braced argument, which it will build a command name from (in this case the name of a variable, in other cases it could be the name of a function). So now we can define the user interface:

```
\NewDocumentCommand{\setvar}{mm}
{
  \example_setvar:cn
    { l_example_var_#1_t1 }
    { #2 }
}
\NewExpandableDocumentCommand{\usevar}{m}
{
  \example_usevar:c { l_example_var_#1_t1 }
}
```

Sites on L^AT_EX are plagued with questions about code doing nasty things such as \def\c{something}, asking why this breaks.

With the approach just outlined we set up a *namespace* for our variables, which can just be called by their ‘outer’ name, leaving to the implementation the details about how to avoid conflicts.

[We could certainly define the user level commands in legacy L^AT_EX. A typical implementation would be

```
\newcommand{\setvar}[2]{%
  \expandafter
    \def\csname example@var@#1\endcsname{#2}%
}
\newcommand{\usevar}[1]{%
  \csname example@var@#1\endcsname
}
```

I won’t quarrel with people maintaining this is simpler. But I’ll remain with my opinion that it isn’t.]

Let’s add something to the game: now we want to allow the user to copy the value of a variable into another, say by doing \copyvar{b}{a}, where b is the new one and a the existing one. We only need a new variant, namely

```
\cs_generate_variant:Nn \example_setvar:Nn
  { cv }
```

and then define the user interface with

```
\NewDocumentCommand{\copyvar}{mm}
{
  \example_setvar:cv
  { l_example_var_#1_t1 }
  { l_example_var_#2_t1 }
}
```

We now have at our disposal another function, namely `\example_setvar:cv`, which takes two braced arguments. The second one is scanned like `c`, producing a symbolic token which should be a variable of some kind and then will deliver *the contents* of the variable as a braced argument to the main function.

[I leave as an easy exercise on `\expandafter` how to do this in legacy L^AT_EX programming (hint: use `\let` and two `\expandafters`, cleverly positioned).]

It's not necessary to write two distinct calls for defining the variants, we can create them both at once with:

```
\cs_generate_variant:Nn
  \example_setvar:Nn
  { cn, cv }
```

The `v` variant is a special case of the `V` variant, which is almost the same, but the capital letter reminds us that a single token (a variable's name) should be used without braces. Suppose we have a function that does something with its argument:

```
\cs_new:Nn \example_foo:n { -- #1 -- }
```

(just some nonsense to illustrate the concept). However, in some cases we need to pass the function something that has been stored in a `t1` variable: nothing simpler, because we can do

```
\cs_generate_variant:Nn \example_foo:n
  { V }
```

allows us to do

```
\example_foo:V \l_tmpa_t1
```

If, say, `\l_tmpa_t1` has been set to contain `abc`, then calling `\example_foo:V \l_tmpa_t1` is exactly the same as doing `\example_foo:n {abc}`.

Bear in mind that variants do not come into existence without first generating them. Kernel functions come predefined with several variants that have proven to be useful; they're listed together with the main function in `interface3.pdf`. What if we don't know whether a variant has already been defined? No problem at all! The generation will be silently ignored and, even if it weren't, there should be no problem either, because variants are generated in a uniform way.

We could avoid generating variants. For instance the job of `\example_setvar:cv` could be accomplished by

```
\exp_args:Ncv \example_setvar:Nn
```

(which is actually how the variant is defined), but there's no point in complicating our life this way. Modern T_EX and friends' implementations have lots of memory available, and the times when memory was in *very* short supply and tricks saving just a few tokens in order to spare memory were necessary are only remembered by old dinosaurs like Frank, Chris, David and myself. I remember well the time I saw the dreaded "This can't happen" error message because I was using P_TC_TE_X.

Let's go back to theory. Are there variants to the function-defining functions? Yes, of course there are! There are times when we want to define a function whose name is decided at runtime. The example below is a bit silly, but should give the idea: the call

```
\cs_new:cn { example_foo:n } { -- #1 -- }
```

is the same as the definition of `\example_foo:n` above, because the name, including the signature, will be formed before the underlying `\cs_new:Nn` function does its job. One can use anything inside the braces corresponding to the `c` argument type, so long as the final result, after *full expansion*, just consists of characters. Oh! Expansion! What is it? Be patient. First there are other bits to discuss.

3.2 Local, global, and *<extra>*s

Everybody should know about *local* and *global*. In L^AT_EX, if we perform some command definition inside an environment, it is local to the environment and will disappear when it ends. Other assignments of meaning are instead global: operations on counters, for instance. We don't need a full discussion on local versus global, but there are aspects of the problem that are relevant for functions.

All `\cs_new<extra>:Nn` declarations are always *global*. Even if performed inside a group, their effect will also be carried on at the outer levels. Further, they will check whether the function is already defined and issue an error message if so. The *<extra>* part will be described next. Variable allocation (not setting) is also always global: the instruction

```
\tl_new:Nn \l_example_foo_t1
```

defines the variable at all levels and will balk if the variable already exists.

However, sometimes we need *locally defined* auxiliary functions, that have no fixed meaning and need to be redefined according to the context. For these there is the family:

```
\cs_set<extra>:Nn
```

The syntax is exactly the same as with `\cs_new`, as are the available variants.

Which one to prefer? The `new` or the `set` family?

The answer is easy: the former, unless the function is *required* to change its definition according to the context. Rarely, if ever, will a high level function be defined with `set`.

Thus, the very nature of the language invites programmers to code in layers. For instance, we could have done:

```
\NewDocumentCommand{\setvar}{mm}
{
  \tl_clear_new:c
  { l_example_var_#1_tl }
  \tl_set:cn
  { l_example_var_#1_tl }
  { #2 }
}
```

without defining `\example_setvar:Nn` at all. I discourage this kind of programming: our code should sit *on top* of `expl3` and provide APIs for the programmer to employ. As I said before, there is no point in sparing a function, even more so if we consider that once our API is available, we can easily define variants thereof for particular jobs.

What's the complete list of `(extra)`s available after `\cs_new` or `\cs_set`? Here it is:

```
\cs_new:Nn
\cs_new_protected:Nn
\cs_new_nopar:Nn
\cs_new_protected_nopar:Nn
\cs_set:Nn
\cs_set_protected:Nn
\cs_set_nopar:Nn
\cs_set_protected_nopar:Nn
\cs_gset:Nn
\cs_gset_protected:Nn
\cs_gset_nopar:Nn
\cs_gset_protected_nopar:Nn
```

In all cases, the first argument is the name of the function to be defined and the second argument is the replacement text.

You probably already have an idea of what `protected` does: it arranges things so that the function is not expanded when *full expansion* is enforced. In particular, a `protected` function cannot be used in a `c`-type argument, because it wouldn't be expanded and it is not a character.

The `nopar` variety disallows `\par` tokens in the function's argument (when called). Unless we're dealing with special situations where `\par` does not make sense in a function's argument, there is no need to use it.

The `gset` family is almost the same as `new`, but no check is performed about the function being defined.

[For the TeX gurus that are reading these notes, `new` and `gset` use `\gdef`, whereas `set` uses `\def`; the `\long` prefix is added except with `nopar`.]

What are the available variants? Here's the complete list for all of the above functions:

`Nn cn Nx cx`

What's this mysterious `x`? It's intended to bring to mind *fully expanded*. There will be a section later on the topic.

3.3 Parameters

Some people will now be complaining that they have seen different ways to define functions and they're right. There *is* a whole new family like the one above but where the signature has a strange `p` between `N` and `n`, namely

```
\cs_new:Npn
\cs_new_protected:Npn
\cs_new_nopar:Npn
\cs_new_protected_nopar:Npn
\cs_set:Npn
\cs_set_protected:Npn
\cs_set_nopar:Npn
\cs_set_protected_nopar:Npn
\cs_gset:Npn
\cs_gset_protected:Npn
\cs_gset_nopar:Npn
\cs_gset_protected_nopar:Npn
```

The `p` is a reserved argument type just for these functions and variants thereof: all of them come along with the variants

`Npn cpn Npx cpx`

and stand for *parameter text*. The two lines below are completely equivalent:

```
\cs_new:Nn \example_usevar:n {...}
\cs_new:Npn \example_usevar:n #1 {...}
```

The same for the other functions. In the second instance, the *parameter text* is explicitly written out. Remember that when the `expl3` programming conventions are in force, spaces are ignored, so for two parameters we can have

```
\cs_new:Nn \example_foo:nn {...}
\cs_new:Npn \example_foo:nn #1 #2 {...}
\cs_new:Npn \example_foo:nn #1#2 {...}
\cs_new:Npn \example_foo:nn #1 #2{...}
\cs_new:Npn \example_foo:nn #1#2{...}
```

and the last four lines are completely equivalent. Personally, I prefer the first way that's 'more logical'; others prefer the second way. Beware! The second way doesn't check for consistency of the signature with the parameter text and it even allows for 'wrong'

signatures, but this fact should not be exploited: L^AT_EX will not balk if you type

```
\cs_new_protected:Npn
  \example_setvar:cn #1 #2
  { ... }
```

but this doesn't mean that you can avoid the two-step procedure of first defining `\example_setvar:Nn` and then creating the variant. Doing the definition this way is *wrong*. The second family of functions even allows for no signature at all, actually, so they can be used for defining user level commands, although the path with `\NewDocumentCommand` (or siblings) is recommended.¹

The p way is necessary when the parameter text is 'nonstandard', in the sense that we're defining a function with delimited arguments; in this case, the signature should be `w`. If we want a function that sets three variables to the year, month and day given an ISO-format date such as 2020-10-15, we can do

```
\int_new:N \l_example_year_int
\int_new:N \l_example_month_int
\int_new:N \l_example_day_int
\cs_new_protected:Nn \example_setdate:n
{
  \__example_setdata:w #1 \q_stop
}
\cs_new_protected:Nn
  \__example_setdate:w #1-#2-#3 \q_stop
{
  \int_set:Nn \l_example_year_int { #1 }
  \int_set:Nn \l_example_month_int { #2 }
  \int_set:Nn \l_example_day_int { #3 }
}
```

Here I introduce another useful convention: if the `<prefix>` is preceded by a double underscore, the function is considered lower level than the others and should *never* be called outside its specific uses by standard functions (without the double underscore). The idea is that the standard functions are the 'programmer's interface', whereas the others are auxiliary whose actual implementation should not concern the programmer. The distinction when writing personal code is not so important, but it is crucial for package writers. Standard functions (without the double underscore) *can* be used by other packages, whereas one should not count on the lower level ones (with the double underscore) to even be defined in later versions of the package.

This should clarify why the code above splits the work into two levels; we have the high level function `\example_setdate:n` function which relies on a lower level one to do the dirty work. Maybe

¹ `expl3` can also be used with plain T_EX, and in this case this is the only way to define user level commands.

the package writer will discover a better way to accomplish the task, but this would only influence the lower level and not the main function, which will be possible to call forever. Maybe the definition of `\example_setdate:n` will change in the future, but this won't affect code that uses it.

4 Expansion

There can be no full understanding of T_EX without some knowledge on how expansion works. In functional programming languages, if `g` is a function of one variable returning an array of three data, whereas `f` is a function of three variables, a call

```
f(g(x))
```

would be permissible (from a mathematical point of view, at least, maybe a particular functional language requires some tweak). This is not the same in T_EX, which goes from outside to inside, rather than conversely.

If we have a one-argument `\example_a:n` function that returns three braced lists of tokens, and another function `\example_b:nnn` that takes three arguments, a call such as

```
\example_b:nnn { \example_a:n {x} }
```

would fail miserably. That's how T_EX works and no clever code can change this. The outer function will look for three arguments and the given braced list of tokens is just one.

As an example of how to handle this, suppose that `\example_a:n` can be fed a date in ISO format and from 2020-10-15 it returns {2020}{10}{15}, whereas `\example_b:nnn` takes three arguments and produces a date in a different format, say "day 'name of the month', year": in this case it should output "15 October, 2020".

We need an indirect approach, in order to allow feeding an ISO date to the general function outputting the date in that format. Let's see how the general function might be defined:

```
\cs_new:Nn \example_date:nnn
  { % #1 = year, #2 = month, #3 = day
    #3~%
    \int_case:nn { #2 }
    {
      {1}{January}
      {2}{February}
      ...
      {12}{December}
    }
    ,~#1
  }
```

The `\int_case:nn` function examines its first argument against the list given as its second argument (the code is incomplete, but you can guess how to

fill it in) consisting of pairs of braced items; the first contains an integer, the second something to output when a match is found. The ~ here is not a nonbreaking space in the `expl3` programming environment, but a normal space.

We also need a function that is given a date in ISO format (2020-10-15) and returns it split into the three constituent parts, {2020}{10}{15}:

```
\cs_new:Nn \__example_isodate:n
{
  \__example_isodate:w #1 \q_stop
}
\cs_new:Npn
  \__example_isodate:w #1-#2-#3 \q_stop
{
  {#1}{#2}{#3}
}
```

The input to the second function is split at the hyphens and at the terminator, so we're using *delimited arguments*, a detail I'll skim over. Here, we just need to know that the call

```
\__example_isodate:n {2020-10-15}
```

will *eventually* return {2020}{10}{15} to the input stream.

How do we combine these? There are several ways, but all of them require understanding the concept of *full expansion*. `TeX` only knows macros; when it finds one, it knows how many arguments it takes and looks for them in the input stream; upon finding them, it replaces the whole sequence of tokens so found with the replacement text of the macro.

The main problem is that most of the time we don't know how many steps of expansion it will take to get from `__example_isodate:n {2020-10-15}` to {2020}{10}{15}. In this case it would be easy to count them, but this is just a simple example. If we knew, a suitable chain of `\expandafter` commands would suffice, but this is prone to errors and inconsistencies if the implementation changes.

What we'd like is for `__example_isodate:n` to go all the way down to the final result in one swoop. Here's a way:

```
\exp_last_unbraced:N
\example_date:nnn
{ \__example_isodate:n {2020-10-15} }
```

What `\exp_last_unbraced` does is *fully expand* its second argument and return the result in the input stream with no braces around it.

There are other ways. One is to define a new helper function:

```
\cs_new:Nn \example_date:n
{
  \__example_date:N
  \example_date:nnn
```

```
{ \__example_isodate:n { #1 } }
}
\cs_new:Nn \__example_date:Nn
{
  #1 #2
}
\cs_generate_variant:Nn
  \__example_date:Nn { Ne }
```

upon which `\example_date:n {2020-10-15}` would produce the intended result.

There are still more ways, but here the idea is to present how we can exploit the full expansion variants.

In sum, there are three kinds of them, namely `e`, `x` and `f`. The last of these is the most restricted, because it performs recursive expansion of the tokens as soon as they are placed in the input stream and ends at the first unexpandable token it finds. Notwithstanding this limitation it has several uses.

The `x` type is nowadays less important because all `TeX` engines supporting `LATEX` have the primitive `\expanded`, which is itself expandable. Only Knuthian `TeX` (the engine that one launches with `tex` on the command line) lacks it, since it is kept with no extensions, according to Knuth's desiderata. [What does `\expanded` do? It is essentially like `\edef`, with the difference that no macro is defined. The argument to it is subject to full recursive expansion which *doesn't* stop when an unexpandable token is found, but just jumps over it and continues from the next token, until exhausting the supplied token list. The result is then placed on the input stream (without braces).]

4.1 Full expansion with e

The `e` argument type tells `LATEX` to first fully expand the given argument and then supply the result to the original function. This is very important if the argument contains a variable which we want to deliver with its value at call time.

We can see an example in a post on `TeX.StackExchange`.² The question is about adding to endnotes, via the `endnote` package, the page number where the endnote actually appears. We want to use `\pageref` through an automatically supplied label:

```
\NewDocumentCommand{\MyEndNote}{m}
{
  \polyv_myendnote:ne
  { #1 }
  { \int_eval:n { \arabic{endnote}+1 } }
}

\cs_new_protected:Nn \polyv_myendnote:nn
{
  \endnote
```

² <https://tex.stackexchange.com/a/438715/>. The code there uses `f`, because `e` wasn't available yet.

```
{
  #1~(page\nobreakspace
  \pageref{\#2:endnote})
}
\label{\arabic{endnote}:endnote}
}
\cs_generate_variant:Nn
  \polyv_myendnote:nn
  { ne }
```

What's the problem to be solved? The endnote number is incremented after the endnote is processed, so a `\label` command gets this new number. So we can't use

```
\pageref{\arabic{endnote}:endnote}
```

because this would refer to the *previous* endnote. Thus the internal function uses `e` expansion in order to generate the successor to the current value of the `endnote` counter. Without this full expansion, all the endnotes declared with `\MyEndNote` would contain the equivalent of

```
\pageref{%
  \int_eval:n{\arabic{endnote}+1}:endnote
}
```

and so the final result would be undefined cross references, because `\arabic{endnote}` would always expand to the *final* value of the counter. For instance, if the last endnote was number 10, we'd end up with `\pageref{11:endnote}`. Instead, with full expansion, the current value is used and passed to the main function. At the first endnote, the counter has value 0, so the end result is the same as

```
\endnote{The text of the endnote
  (page\nobreakspace\pageref{1:endnote})}
\label{1:endnote}
```

We could as well have used `f` or `x` for this particular application, but `e` is the most efficient of the lot. The difference from `x` is that functions using `x` are *not* expandable, so they have to be of the `protected` kind. Indeed the process is a two-step one: first a temporary token list is set using

```
\tl_set:Nx \l__exp_internal_t1 {...}
```

(which internally uses good old `\edef`).

The introduction of `e`-type full expansion has been a significant step forward, because it allows for things that were almost impossible before.

However, as seen above, there are uses for `x`: for instance there is no `\cs_new:Ne` variant and it would be *less* efficient than `\cs_new:Nx` (which is just `\edef`).

5 Another essential variant: V for variables

In the list of argument types above there are `V` and `v`, which have been touched upon briefly. Now it's time to discuss the former in greater detail.

Type `v` is nearly the same as `V`; it just adds the ability of building the name of the variable by supplying data at runtime. The main one is `V`.

Again, let's suppose we have our favorite function that splits an ISO date into components and outputs the date in another format, and that it's named `\example_date:n`. (Its implementation is irrelevant.)

Suppose now we have a date stored in a `t1` variable. Since this is just a macro under cover, at the beginning of `expl3`, the way to process this was

```
\cs_generate_variant:Nn
  \example_date:n { o }
```

to be called like

```
\example_date:o { \l_tmpa_t1 }
```

but this is bad because it depends on the knowledge of the implementation of `t1` variables. Also, it is not generalizable to other kinds of variables. The `o` variants do a single expansion step in the braced argument; while this works with the straightforward implementation of token list variables it would fail spectacularly with `fp` variables (which contain floating point numbers).

The best method is to do

```
\cs_generate_variant:Nn
  \example_date:n { V }
```

to be called like

```
\example_date:V \l_tmpa_t1
```

This will deliver the contents of the variable, suitably braced, to the main function. So if we did

```
\tl_set:Nn \l_tmpa_t1 { 2020-10-15 }
```

then the call `\example_date:V \l_tmpa_t1` would be equivalent to `\example_date:n {2020-10-15}`.

An example with a different kind of variable, namely `int`. We want to feed in such a variable and the result should pad it with zeros to get four digits:

```
\cs_new:Nn \example_pad:n
{
  \prg_replicate:nn
    { 4 - \tl_count:n { #1 } }
    { 0 }
  #1
}
\cs_generate_variant:Nn
  \example_pad:n { V }

\int_set:Nn \l_tmpa_int { 43 }
\example_pad:V \l_tmpa_int
```

This will print 0043. This may seem of academic interest only, but with the sibling `v` type, we can transform this into

```
\cs_generate_variant:Nn
  \example_pad:n { v }
\example_pad:v { c@page }
```

knowing that `\c@page` is the L^AT_EX name of the register containing the page number and an application can be immediately thought of. This exploits the fact that standard T_EX counters are exactly like `expl3` int variables. Using the V or v variants we're passing the main function the actual value as a list of digits, so we can count it, which would be impossible with the ‘abstract value’.

What variable types can be used this way? Several: `t1`, `int`, `fp`; also `clist` and others more esoteric. Essentially, all variables that can deliver some sensible output. This cannot be expected from `seq` or `prop` variables and, indeed, using those will crash.

I've sometimes found it useful to define the `\cs_set:NV` variant for `\cs_set:Nn` in order to use a `t1` variable where the desired replacement text has been stored and modified via some regular expression replacement.³ By the way, it is not possible to have a `\cs_set:NpV` variant, because the generator `\cs_generate_variant:Nn` can only accept a function with a signature consisting of N or n characters.

6 True or false?

There are two other interesting argument types: T and F, and the title of the section should suggest that they're connected with truth and falsehood.

Exactly so! They are argument specifiers in the signature of *conditional* functions. Example:

```
\int_compare:nTF
```

is a function that takes three standard braced arguments; the first is a numeric relation between integers to test, the second the code to execute if the relation is true, the third the code to execute if the relation is false. So

```
\int_compare:nTF { 0<1 } { A } { B }
\int_compare:nTF { 0>1 } { A } { B }
```

will result in printing A and B respectively. So, why isn't it more simply `\int_compare:nnn`? Indeed, it used to be this way in the first versions of `expl3`, but it was realized that having different argument specifiers is handier, because we can also have

```
\int_compare:nT
\int_compare:nF
```

³ <https://tex.stackexchange.com/a/355576/>

when we have nothing to execute for the false or true branch respectively. Of course

```
\int_compare:nF { <relation> } { B }
\int_compare:nTF { <relation> } { } { B }
```

are completely equivalent, but the former shows more clearly that we want to do nothing if the `<relation>` turns out to be true. With the :nnn signature, empty arguments would be always required. Also, the presence of either T or F (or both) immediately alerts us that the function is a conditional.

All kernel conditional functions are available with ending TF, T or F; some conditional-like functions even have the version with neither. For instance we can see in `interface3.pdf` that there is `\str_case:nn`, but also `\str_case:nnTF`. The strange-looking `TF` means that all three combinations TF, T and F available.

Why such a “pseudo-conditional”? The function `\str_case:nn` is not a conditional (being related to letter case), but we can use the extended version to output something if there is, or is not, a match. The version which is most likely to be used is `\str_case:nnF` to output something in case of no match, maybe an error message or a default output.

To generate proper conditionals, variants should be defined with

```
\prg_generate_conditional_variant:Nnn
```

rather than with `\cs_generate_variant:Nn`. For instance, if we plan to store some `<relations>`s for `\int_compare:nTF` into `t1` variables, the correct way to generate the variant is

```
\prg_generate_conditional_variant:Nnn
  \int_compare:n { V } { p, TF, T, F }
```

This will at once generate the variants

```
\int_compare:VTF
\int_compare:VT
\int_compare:VF
```

as well as the ‘predicate form’

```
\int_compare_p:V
```

to be used in boolean expressions. But this is outside the scope of the present paper.

◇ Enrico Gregorio
Dipartimento di Informatica,
Università di Verona
enrico dot gregorio (at) univr
dot it

bib2gls: selection, cross-references and locations

Nicola L. C. Talbot

Abstract

In my previous article [6], I described using indexing applications with L^AT_EX, a process required by the glossaries package to sort and collate terms, and the development of the `bib2gls` command line application, which was designed specifically for the glossaries-extra extension package. This article describes how `bib2gls` differs from the other indexing methods with respect to selection, grouping, cross-references and invisible locations.

1 \printglossary vs \printunsrtglossary

In order to better understand how items are listed with `bib2gls` [3], it's useful to understand the principal differences between `\printglossary` (which is provided by `glossaries` [4] and used with `makeindex` and `xindy` [1]) and `\printunsrtglossary` (which is provided by `glossaries-extra` [5] and used with `bib2gls`). This was briefly covered in the previous article but is described in more detail here.

Consider the following document:

```
\documentclass{article}
\usepackage[style=treegroup]{glossaries}
\makeglossaries
\loadglsentries{entries}
\begin{document}
\Gls{duck}, \gls{parrot} and \gls{quartz}.
\printglossary
\end{document}
```

The entries are all defined in the file `entries.tex`, which helps reduce clutter in the main document file and also makes it easier to reuse the same definitions in other documents. The contents of this file follows:

```
\newglossaryentry{antigen}{name={antigen},
  description={toxin or other foreign substance
    that induces an immune response}}
\newglossaryentry{mineral}{name={mineral},
  description={solid, inorganic,
    naturally-occurring substance}}
\newglossaryentry{animal}{name={animal},
  description={living organism that has
    specialised sense organs and nervous system}}
\newglossaryentry{bird}{name={bird},
  parent={animal},
  description={egg-laying animal with feathers,
    wings and a beak}}
\newglossaryentry{parrot}{name={parrot},
  parent={bird},
  description={mainly tropical bird with bright
    plumage}}
\newglossaryentry{duck}{name={duck},
```

```
parent={bird},
description={waterbird with webbed feet}}
\newglossaryentry{quartz}{name={quartz},
parent={mineral},
description={hard mineral consisting of silica}}
```

This defines seven glossary entries. Only three have been referenced in the document, three are ancestors of the referenced entries so they must be included in the glossary as well, and one (`antigen`) hasn't been referenced and isn't required by any referenced entry. The document build is:¹

```
latex myDoc
makeglossaries myDoc
latex myDoc
```

(assuming the document source is in the file `myDoc.tex`). The `makeglossaries` helper script invokes `makeindex`, which creates the file `myDoc.gls` that contains (line breaks added for clarity throughout):

```
\glossarysection[\glossarytoctitle]
  {\glossarytitle}
\glossarypreamble
\begin{theglossary}\glossaryheader
\glsgroupheading{A}\relax \reset
\glossentry{animal}\relax \reset
\subglossentry{1}{bird}\relax \reset
\subglossentry{2}{duck}\{\locationlist\}
\subglossentry{2}{parrot}\{\locationlist\}
\glsgroupskip
\glsgroupheading{M}\relax \reset
\glossentry{mineral}\relax \reset
\subglossentry{1}{quartz}\{\locationlist\}
\end{theglossary}\glossarypostamble
```

(The `\reset` code, which is omitted for clarity, deals with counteracting the effect of `\glsnonextpages`.) Note that the location list argument for the unreferenced ancestor entries is just `\relax`. The start of each letter group is identified with

```
\glsgroupheading{\grouplabel}
```

The argument is a label which may have a corresponding title. If there's no title associated with it the label is used as the title. Glossary styles that don't support group headings define this command to do nothing.

`\printglossary` effectively does:

```
\begin{bgroup}
  \processoptions
  \ifinputglossaryfile{input glossary file if it exists}
\egroup
```

The initialisation parts (`\setupdefaults` and `\processoptions`) deal with defining the glossary section title (`\glossarytitle` and `\glossarytoctitle`), the

¹ `latex` is used here to denote `pdflatex`, `xelatex` or `lualatex`. Replace as appropriate.

preamble and postamble, and implementing the required glossary style (which defines `theglossary` and the formatting commands used in that environment).

A few minor modifications are needed to the example document to use `\printunsrtglossary` instead:

```
\documentclass{article}
\usepackage[postdot,stylemods,style=treegroup]
    {glossaries-extra}
\loadglsentries{entries}
\begin{document}
\Gls{duck}, \gls{parrot} and \gls{quartz}.
\printunsrtglossary
\end{document}
```

Note that `\makeglossaries` has been removed as there are now no indexing files that need to be opened. The extension package has a different set of defaults to the base package, so the post-description punctuation needs to be added (`postdot`) if required. The `stylemods` option automatically loads `glossaries-extra-stylemods` which modifies the predefined glossary styles to provide better integration with `glossaries-extra` and `bib2gls` and to make the styles easier to customise.

The document build is now simply:

```
latex myDoc
```

In this case there's no file for `\printunsrtglossary` to input. Instead, it iterates over all defined entries for the given glossary to obtain the contents. Some glossary styles use a tabular-like environment and loops within such environments are problematic, so an internal control sequence (`\@glsxtr@doglossary`) is used to store the contents of the glossary which is then expanded on completion. The glossary code is now essentially:

```
setup defaults
\begin{group}
process options
\glossarysection[\glossarytoctitle]
    {\glossarytitle}
\glossarypreamble
\@glsxtr@doglossary>
\printunsrtglossarypredoglossary
\@glsxtr@doglossary
\glossarypostamble
\end{group}
```

The `\@glsxtr@doglossary` command ends up defined as:

```
\begin{theglossary}\glossaryheader reset
content
\end{theglossary}
```

The `\iotacontent` part is constructed within a loop. The current group label is initialised to empty:

```
\def\@gls@currentlettergroup{}
```

Each iteration of the loop performs the following steps:

1. Do the loop hook (which does nothing by default but may be configured to skip the current entry).
2. If the current entry doesn't have a parent, obtain its group label (empty, if unavailable), and if the `\iotagroup label` for this entry is different from the currently stored group label then add the following code to `\iotacontent`:
`\glsgroupheading{\group label}`
 (if the current group label is empty) or
`\glsgroupskip\glsgroupheading{\group label}`
 (if the current group label isn't empty). The current group label is then set to `\iotagroup label`.
3. Add the following to `\iotacontent`:
`\internalcs{handler}{\entry label}`

The group label is obtained as follows: if the `group` key has been defined then the label is obtained from the entry's `group` field (which may be empty) otherwise the label is obtained from the uppercase character code of the first letter of the `sort` field (which is normally obtained from the `name` field if not set).

In this example, the entry on the first iteration of the loop is 'antigen'. This entry doesn't have a parent so the group information is queried to determine if a new group heading should be inserted.

The `group` key hasn't been defined in this document, so the group label needs to be obtained from the first character of the `name` field (since the `sort` field hasn't been provided). This character is the letter 'a' so the label is set to the decimal code of its uppercase equivalent (65). This is different from the current group label (initially empty), so the group header command is added:

```
\glsgroupheading{65}
```

(The decimal code is used for the group label to make it easier to expand.)

Note that no `\glsgroupskip` is added at this point because the current group label was empty. The new current group label is updated (to 65). The internal handler macro is then added:

```
\@printunsrt@glossary@handler{antigen}
```

This handler macro is used by all entries, regardless of their hierarchical level, and it uses the command:

```
\printunsrtglossaryhandler{\label}
```

This is the command that should be redefined (not the internal handler macro) if you want to customize the output. The default definition is simply

```
\glsxtrunsrtdef{\label}
```

This fetches the entry's hierarchical level and then does either ($\langle level \rangle = 0$)

```
\glossentry{\label}{\location}
```

or ($\langle level \rangle > 0$)

```
\subglossentry{\label}{\location}
```

where the location list is obtained from an internal field. In this example those fields haven't been set, so the locations are all empty.

For debugging purposes, it's possible to see the glossary code content using:

```
\renewcommand{\printunsrtglossary}{%
\csshow{@glsxtr@glossary}}
```

In the above example, the content is:

```
\begin{theglossary}\glossaryheader {\reset}
\glsgroupheading{65}
\@printunsrt@glossary@handler{antigen}
\glsgroupskip\glsgroupheading{77}
\@printunsrt@glossary@handler{mineral}
\glsgroupskip\glsgroupheading{65}
\@printunsrt@glossary@handler{animal}
\@printunsrt@glossary@handler{bird}
\@printunsrt@glossary@handler{parrot}
\@printunsrt@glossary@handler{duck}
\@printunsrt@glossary@handler{quartz}
\end{theglossary}
```

(There's only one $\langle reset \rangle$ here as there's no sense in using \glsnonextpages with \printunsrtglossary .)

The results of both methods are shown in Figures 1 and 2. Note that the letter groups show the decimal character code (used as the group label) because no title has been assigned. Titles may be assigned with

```
\glsxtrsetgroup{label}{title}
```

For example:

```
\glsxtrsetgroup{65}{A}
```

Obviously this is quite tedious to do for the entire alphabet.

The order of definition has created some strange results: there are two groups with the label 65 ('A') and the 'quartz' sub-entry is separated from its parent ('mineral'). The glossary style determines whether or not the hierarchy is visible (through indentation etc.). The internal loop doesn't make any attempt to gather child entries. The `parent` field is only queried within the loop to determine whether or not to attempt to insert the letter group headings.

The key to using \printunsrtglossary is to ensure the entries are defined in the correct order, defining child entries immediately after their parent, and defining only those entries which are required. In this case, the entries should be defined in the order: animal, bird, duck, parrot, mineral, quartz (antigen shouldn't be defined as it's not required).

The way `bib2gls` works is by fetching data from `.bib` files and creating a file (`.glstex`) that defines all required entries in the required order with the required internal fields (for the group label and location lists) set appropriately. Wrapper commands are provided to make it easier to customise. For example:

```
\providecommand{\bibglsnewentry}[4]{%
\longnewglossaryentry*{\#1}{name=\#3,\#2}{\#4}}%
(\longnewglossaryentry* is used to allow for multi-paragraph descriptions.)
```

If required, the group labels are obtained by the sort method and the code to define the corresponding titles is added to the `.glstex` file. In other words, `bib2gls` takes care of all the tedious code that's required with the manual method. This behaviour is possible to override; however, if `bib2gls` is instructed to assign group labels that don't follow the order obtained by the given sorting method then fragmented groups will occur. (If you find yourself wanting to order by group title then this is an indication that you should actually be using a hierarchical system instead [7].)

The `.glstex` file is input (if it exists) with:

```
\GlsXtrLoadResources[options]
```

This command also writes information to the `.aux` file that's picked up by `bib2gls` (for example, the names of the `.bib` files that contain the data and how to order the entries).

`bib2gls` comes with a helper command line utility `convertgls2bib` which can be used to parse `TEX` files for instances of `\newglossaryentry` and other commands that are provided to define entries (such as `\newabbreviation`). In general, it's best to use this tool with files that only contain entry definitions (such as the example `entries.tex`) but it can also be used on a complete document. (In this case, the `-p` or `--preamble-only` switch may be used to limit parsing to the document preamble.) For example:

```
convertgls2bib entries.tex entries.bib
```

This will create a file called `entries.bib`. The example `myDoc.tex` file can now be modified to use `bib2gls`:

```
\documentclass{article}
\usepackage[record,postdot,style=treegroup]
  {glossaries-extra}
\GlsXtrLoadResources[src={entries}]
\begin{document}
\Gls{duck}, \gls{parrot} and \gls{quartz}.
\printunsrtglossary
\end{document}
```

Note the use of the `record` package option, which is required with `bib2gls`. This option defines the `group` key, which defaults to an empty label if not

explicitly assigned, and the `location` key, which is used to store the formatted location list (another field is available that stores each location in an internal list, if required).

The document build is now:

```
latex myDoc
bib2gls -g myDoc
latex myDoc
```

The result is shown in Figure 3.

The `-g` (or `--group`) switch is required if you want distinct groups. This will make the sort methods automatically assign the group label to each top-level entry (stored in the entry's `group` field). If this switch isn't used and the group labels aren't assigned in some other way, then step 2 in the loop iteration (page 309) will be skipped.

Note there's a difference between using the `-g` switch with a style that doesn't show the group title and not using the `-g` switch. For example, if the style is changed from `treegroup` to `tree` then when `bib2gls` is invoked with `-g` there will be a vertical gap between letter groups (unless the `nogroupskip` option is used) whereas there won't be a gap if `bib2gls` is run with the default `--no-group` setting.

In the first case, the group label is set, so step 2 in the loop iteration adds the group skip and group heading commands. The `tree` style redefines the group heading command to do nothing but the group skip is implemented. In the second case, the group label isn't set, so step 2 is omitted, so neither the group skip nor the group heading command will be inserted. If the `nogroupskip` option is set with a glossary style that doesn't show the group heading, then the result will typically appear the same as invoking `bib2gls` with the default `--no-group` setting. However, since the group formations add to the total document build time it's more efficient to simply use the default `--no-group` setting—unless you have multiple glossaries where some do require visual separation between groups.

2 The .bib file

As with BIBTeX, data is defined in the `.bib` file in the form:

```
@<entry-type>{<label>, <key=value list>}
```

If the `<entry-type>` is unrecognised, it will be ignored (with a warning). Comments are slightly different: in BIBTeX, anything outside of `@<entry-type>{...}` is considered a comment, but `bib2gls` is stricter and comments need to be marked up as such. Like TeX, `bib2gls` recognises `%` as a comment character. The most important comment is the encoding line, e.g.:

```
% Encoding: UTF-8
```

This is best placed near the start of the file. General comments (but not the encoding) may also be supplied in `@comment`. For example:

```
@Comment{jabref-meta: databaseType:bib2gls;}
```

(Entry type names are case-insensitive.) There are four basic sets of entry types:

abbreviations Two primary entry types:

`@abbreviation` and `@acronym`. These have two required fields: `short` and `long`.

symbols Two primary entry types: `@symbol` and `@number`. The required fields are: `name` or `parent`. If the `name` is missing, then the `description` is also required.

index Two primary entry types: `@index` and `@indexplural`. There are no required fields.

general One primary entry type: `@entry`. The required fields are: `description` and either `name` or `parent`.

There are other entry types, but they are beyond the scope of this article.

Unknown entry types and fields can be aliased, which can make a `.bib` file more adaptable to multiple documents. For example, consider:

```
@unit{m,
  unitname={metre},
  unitsymbol={\si{\metre}},
  measurement={length}
}
```

This is an unknown entry type where all the fields are also unknown. However, the resource options

```
entry-type-aliases={unit=entry},
field-aliases={
  unitname=name,
  unitsymbol=symbol,
  measurement=description
}
```

will make `bib2gls` treat this entry as though it had been defined as

```
@entry{m,
  name={metre},
  symbol={\si{\metre}},
  description={length}
}
```

whereas

```
entry-type-aliases={unit=symbol},
field-aliases={
  unitname=description,
  unitsymbol=name
}
```

will make `bib2gls` treat this entry as though it had been defined as

```
@symbol{m,
```

Glossary

A

animal living organism that has specialised sense organs and nervous system.

bird egg-laying animal with feathers, wings and a beak.

duck waterbird with webbed feet. 1

parrot mainly tropical bird with bright plumage. 1

M

mineral solid, inorganic, naturally-occurring substance.

quartz hard mineral consisting of silica. 1

Figure 1: \printglossary (ordered by makeindex)

Glossary

65

antigen toxin or other foreign substance that induces an immune response.

77

mineral solid, inorganic, naturally-occurring substance.

65

animal living organism that has specialised sense organs and nervous system.

bird egg-laying animal with feathers, wings and a beak.

parrot mainly tropical bird with bright plumage.

duck waterbird with webbed feet.

quartz hard mineral consisting of silica.

Figure 2: \printunsrtglossary and stylemods (no automated ordering)

Glossary

A

animal living organism that has specialised sense organs and nervous system.

bird egg-laying animal with feathers, wings and a beak.

duck waterbird with webbed feet. 1

parrot mainly tropical bird with bright plumage. 1

M

mineral solid, inorganic, naturally-occurring substance.

quartz hard mineral consisting of silica. 1

Figure 3: \printunsrtglossary and stylemods (ordered with bib2gls --group)

```

description={metre},
name={\sif{\metre}}
}

```

With the other indexing options (`makeindex`, `xindy` or `\printnoidxglossary`), the general recommendation is to set the `sort` key for any entry that contains commands within the `name`. For example:

```
\newglossaryentry{m}{name={\sif{\metre}},
sort={m},description={metre}}
```

With `bib2gls`, the recommendation is the opposite: the `sort` field typically *shouldn't* be set [8]. For this reason, by default `convertgls2bib` will skip the `sort` field when parsing commands like `\newglossaryentry`. By omitting this field, it becomes possible to dynamically allocate the most appropriate value on a per-document basis, which makes it much easier to share `.bib` files across multiple documents. This will be covered in more detail in a follow-up article.

3 Cross-referencing

When using `\index` with `makeindex`, if you want to add a cross-reference in the index then you use the `see` or `seealso` encap (format). For example:

```
\index{cross product|see{vector product}}
\index{dot product|seealso{vector product}}
\index{products|see{dot product and vector
product}}
```

These are treated by `makeindex` in the same way as any other location format, where the content following the encap marker (the vertical pipe | by default) is treated as the name of a formatting command that needs to encapsulate the page number. The argument text `{vector product}` is considered all part of the formatting command name (from `makeindex`'s point of view). The above commands will be converted by `makeindex` into:

```
\item cross product, \see{vector product}{1}
...
\item dot product, \seealso{vector product}{1}
...
\item products, \see{dot product and vector
product}{1}
```

(assuming the `\index` commands were on page 1). The `\see` and `\seealso` commands are provided by indexing packages such as `makeidx` [2] and are defined to ignore the second argument. Naturally, you also need to index the referenced term ('vector product' in this case) to avoid confusing the reader.

By analogy, you could adopt the same method with the `glossaries` package (`makeidx` is loaded in the example below to provide `\see` and `\seealso`):

```

\documentclass{article}
\usepackage{makeidx}
\usepackage{glossaries}
\makeglossaries
\newglossaryentry{product}{name={products},
description={...}}
\newglossaryentry{vector-product}{
name={vector product},description={...}}
\newglossaryentry{cross-product}{
name={cross product},description={...}}
\newglossaryentry{dot-product}{
name={dot product},description={...}}
\begin{document}
\Gls{vector-product}.
\glsadd[format=see{vector product}]
{cross-product}
\glsadd[format=seealso{vector product}]
{dot-product}
\glsadd[format=see{dot product and vector
product}]{product}
\printglossaries
\end{document}

```

In version 1.17 (2008-12-26) of the base `glossaries` package a new command `\glssee` was added to provide a cross-referenced entry similar to this, but instead of using `makeidx`'s `\see` and `\seealso` commands it uses its own analogous commands that take a label as the first argument instead of user-supplied text. (Again the second argument containing the location is ignored.) So the above document can be changed to use `\glssee`:

```

\documentclass{article}
\usepackage{glossaries}
\makeglossaries
\newglossaryentry{product}{name={products},
description={...}}
\newglossaryentry{vector-product}{
name={vector product},description={...}}
\newglossaryentry{cross-product}{
name={cross product},description={...}}
\newglossaryentry{dot-product}{
name={dot product},description={...}}
\begin{document}
\Gls{vector-product}.
\glssee{cross-product}{vector-product}
\glssee[see also]{dot-product}{vector-product}
\glssee{product}{dot-product,vector-product}
\printglossaries
\end{document}

```

This has several advantages:

- the cross-references are identified by label so the text produced can be obtained from the `name` key, which ensures consistency;
- if the `hyperref` package is added then the cross-reference can be automatically hyperlinked;

- if `xindy` is required instead of `makeindex`, then `\glssee` can use `xindy`'s native cross-referencing markup.

The location (which is ignored within the document but required by `makeindex`) is set to 'Z' regardless of where `\glssee` is used in the document so, with the default `makeindex` settings, the cross-reference will be pushed to the end of the location list.

In the case of synonyms, such as 'cross product', that don't need to be used in the document but need to be added to the glossary as a cross-reference to assist the reader, then the term only needs to be defined and indexed with `\glssee`. For convenience, version 1.17 also introduced the `see` key to `\newglossaryentry` as a shortcut to enable the entry to be defined and indexed at the same time. For example:

```
\newglossaryentry{cross-product}{
  name={cross product},description={...},
  see={vector-product}}
```

is equivalent to:

```
\newglossaryentry{cross-product}{
  name={cross product},description={...}}
\glssee{cross-product}{vector-product}
```

This is the only function that the `see` key serves with the base `glossaries` package. Since indexing can only be performed after the associated files have been opened an error will occur if the `see` key is used before `\makeglossaries` (otherwise the indexing will silently fail). For draft documents (where you may want to consider commenting out `\makeglossaries` to speed compilation), you can suppress the error or turn it into a warning with the `seenoindex` package option.

As with `\index`, it's necessary to ensure that the referenced entry is also indexed (through commands like `\gls` or `\glsadd`).

The `glossaries-extra` package provides a similar command `\glsxtrindexseealso`, which essentially does `\glssee[\seesealsoname]` (unless `xindy` is required, in which case alternative markup is used). There's a corresponding key `sealso` that performs this command, analogous to the `see` key. (Note that the tag used for the 'see also' command and key is always `\seesealsoname`.) Although these commands (and their corresponding shortcut keys) essentially do the same thing but with a different tag, they are provided both for semantic reasons and to make it easier to apply different formatting, depending on whether the cross-reference is a synonym or a pointer to related terms.

The extension package modifies the `see` key so that its value is also saved. The key still serves

as a shortcut for `\glssee`, but it may be useful to later query the information. The `sealso` key also saves its value. The extension package also provides a related key `alias` which may only take a single label as its value. This behaves much like its `see` counterpart when indexing but it will also make commands like `\gls` link to the alias target in the glossary.

Now let's switch to `\printunsrtglossary`:

```
\documentclass{article}
\usepackage{hyperref}
\usepackage[seenoindex=ignore]{glossaries-extra}
\newglossaryentry{product}{name={products},
  see={dot-product,vector-product},
  description={...}}
\newglossaryentry{vector-product}{name={vector product},description={...}}
\newglossaryentry{cross-product}{name={cross product},description={...},
  alias={vector-product}}
\newglossaryentry{dot-product}{name={dot product},description={...},
  sealso={vector-product}}
\begin{document}
\Gls{vector-product} (also called
\gls{cross-product}) and \gls{dot-product}.
\printunsrtglossary
\end{document}
```

No indexing is performed so the `see` and `sealso` keys have no effect. There are no location lists for any of the entries (not even the ones used in the document). In order to show the cross-referencing information in the glossary, it's necessary to either modify the glossary style (or associated hooks) or define the `location` key (which the `record` option does) and then set this key for the required entries. For example:

```
\usepackage[record]{glossaries-extra}
\newglossaryentry{product}{name={products},
  see={dot-product,vector-product},
  description={...},
  location={\glsxtrusesee{product}}}
\newglossaryentry{vector-product}{name={vector product},description={...}}
\newglossaryentry{cross-product}{name={cross product},description={...},
  alias={vector-product},
  location={\glsxtrusealias{cross-product}}}
\newglossaryentry{dot-product}{name={dot product},description={...},
  sealso={vector-product},
  location={\glsxtruseseealso{dot-product}}}
```

Again this is tedious to do manually but can be performed automatically by `bib2gls`.

In `.bib` files, the `see`, `sealso` and `alias` fields don't perform any automated indexing but establish

dependencies. The entries that are actually selected and added to the `.glstex` file depend on the selection criteria. For example:

```
\usepackage[record]{glossaries-extra}
\GlsXtrLoadResources[src=entries,selection=all]
\begin{document}
\printunsrtglossaries
\end{document}
```

This will select all entries defined in `entries.bib`. None of them will have any page numbers (because they haven't been indexed in the document), but any entries with the `see`, `seealso` or `alias` fields set will have the cross-reference information added to the `location` field.

The default setting is `selection={recorded and deps}` which selects all entries with records in the `.aux` file (that is, they've been indexed using commands like `\gls`) and their dependent entries (ancestors, cross-references and any entries that have been referenced in certain fields, such as `description`). This is straightforward for `bib2gls` to do (since it has access to all data in the `.bib` files) but is something that `makeindex` and `xindy` can't do (as they only have limited information about entries that have been indexed and no information at all about entries that haven't been indexed).

Consider the following example (which requires `makeindex`):

```
\documentclass{article}
\usepackage[colorlinks]{hyperref}
\usepackage[style=tree]{glossaries-extra}
\makeglossaries
\loadglseentries{vegetables}
\begin{document}
\Gls{cauliflower} and \gls{marrow}.
\printglossaries
\end{document}
```

Where the file `vegetables.tex` contains:

```
\newglossaryentry{cauliflower}{
    name={cauliflower},description={type of
        \gls{cabbage} with edible white flower head}}
\newglossaryentry{cabbage}{
    name={cabbage},description={vegetable
        with thick green or purple leaves}}
\newglossaryentry{marrow}{
    name={marrow},description={long
        white-fleshed gourd with green skin},
   seealso={courgette}}
\newglossaryentry{courgette}{name={courgette},
    description={immature fruit of a \gls{marrow}}}
\newglossaryentry{zucchini}{name={zucchini},
    description={} , see={courgette}}
\newglossaryentry{aubergine}{name={aubergine},
    description={purple egg-shaped fruit}}
\newglossaryentry{eggplant}{name={eggplant},
    description={} , see={aubergine}}
```

Two entries have been indexed in the document (cauliflower and marrow) and three have been implicitly indexed via the `see` or `seealso` key (marrow, zucchini and eggplant). If the file is called `myDoc.tex` then the document build would normally be:

```
latex myDoc
makeglossaries myDoc
latex myDoc
```

This results in a glossary containing five items (cauliflower, courgette, eggplant, marrow and zucchini; see Figure 4), and there are two warnings from `hyperref` about non-existent references to targets `glo:aubergine` and `glo:cabbage`. This is because there are hyperlinks in the glossary to aubergine and cabbage, but the targets aren't defined as those entries haven't been indexed. In the case of cabbage, `makeindex` isn't aware of the reference in the description of cauliflower, but once the glossary has been created this reference can be indexed on the next L^AT_EX run. This means that the complete document build has to be:

```
latex myDoc
makeglossaries myDoc
latex myDoc
makeglossaries myDoc
latex myDoc
```

This ensures that the required cabbage entry appears in the glossary but there's still a broken link to the unlisted aubergine (Figure 5). The cross-reference (via `see` or `\glssee`) only indexes the source entry (eggplant). It doesn't index the target (aubergine). The target must be indexed in order to resolve the broken link, but there's no reason for either eggplant or aubergine to be listed in the glossary as neither are required in the document.

The `vegetables.tex` file can be converted to a `.bib` file:

```
convertgls2bib -i vegetables.tex vegetables.bib
```

(The `-i` switch converts the entries with empty descriptions to use `@index` instead of `@entry`, which is more appropriate.) The document can now be converted to use `bib2gls`:

```
\documentclass{article}
\usepackage[colorlinks]{hyperref}
\usepackage[record,stylemods,style=tree]
    {glossaries-extra}
\GlsXtrLoadResources[src={vegetables}]
\begin{document}
\Gls{cauliflower} and \gls{marrow}.
\printunsrtglossaries
\end{document}
```

This is with the default selection criteria which selects recorded entries (cauliflower and marrow) and their dependencies (cauliflower requires cabbage, since

`\gls{cabbage}` is in its description, and marrow requires courgette, in order to resolve the cross-reference). This means that the glossary ends up with four items: cabbage, cauliflower, courgette and marrow. Note that cabbage doesn't have a location. The location (if required) can only be determined once the description is expanded in the glossary.

Neither zucchini nor eggplant have been selected since neither of them have records and neither are required by any of the indexed entries (or their dependents). It would, however, be useful to also select zucchini to supply the synonym for courgette (but not eggplant, since aubergine isn't required). This can be done with either `selection={recorded and deps and see}` or `selection={recorded and deps and see not also}`. This will select any entries that cross-reference a required entry via the `see` or `alias` fields. The former will also include cross-references via the `seealso` field. The latter doesn't. This will now include zucchini but not eggplant (Figure 7).

So with `bib2gls` you can use `see`, `seealso` and `alias` to establish dependencies without automatically forcing the entry into the glossary. With the other methods, these keys should only be used if that automated indexing is intended.

4 Invisible or ignored locations

Both `makeindex` and `xindy` require an associated location (typically a page number). They are general purpose indexing applications and indexes are intended to direct the reader to relevant locations in the document. Glossaries, on the other hand, provide definitions of terms and these don't necessarily require any locations. The location list may be suppressed with the `nonumberlist` option, but this will also suppress any cross-references (since they are placed inside the location list).

The `glossaries` package provides a `\@gobble`-like command `\glsignore` which simply ignores its argument and may be used as an encap to provide an invisible location. This only works if that is the only location in the list. If there are other locations it will result in spurious commas or en-dashes. This encap is used by `\glsaddallunused`, which iterates over all defined entries and indexes each unused entry. The aim here is to ensure all entries appear in the glossary, while only those used in the text have locations. The problematic spurious commas and en-dashes occur when this command is combined with any indexing command that doesn't mark the entry as used or if the first-use flag has been reset or if any subsequent indexing occurs.

Since `bib2gls` is designed for glossaries where locations may not be required, it allows selection without adding to the location list. The `bib2gls` alternative to `\glsaddallunused` is to use `selection=all`, which will select all entries, but only those that have been specifically indexed will have locations. It also recognises `glsignore` as a special 'ignored location', which indicates that the entry should be selected but the location should be discarded (rather than simply rendered invisible). You can even set this as the default format with

`\GlsXtrSetDefaultNumberFormat{glsignore}`

This could, for example, be done at the start of the back matter, or it could be done for the entire document and only overridden for significant locations. Setting up the alternative modifier can make it easier to switch the format. For example:

`\GlsXtrSetAltModifier{!}{format=glsnumberformat}`

Now the principal mention of cauliflower could be written as:

A `\gls!{cauliflower}` is a type of `\gls{cabbage}`.

If `glsignore` has been set as the default format this will only add the current page to the cauliflower location list but will ensure that cabbage is also selected. This can help reduce lengthy location lists into a more compact list that only includes the most pertinent locations.

References

- [1] R. Kehr, J. Schrod. `xindy`: a general-purpose index processor, 2018. ctan.org/pkg/xindy.
- [2] L^AT_EX Team. The `makeidx` package, 2014. ctan.org/pkg/makeidx.
- [3] N. Talbot. `bib2gls`: Command line application to convert `.bib` files to `glossaries-extra.sty` resource files, 2020. ctan.org/pkg/bib2gls.
- [4] N. Talbot. The `glossaries` package, 2020. ctan.org/pkg/glossaries.
- [5] N. Talbot. The `glossaries-extra` package, 2020. ctan.org/pkg/glossaries-extra.
- [6] N. Talbot. Indexing, glossaries and `bib2gls`. *TUGboat* 40(1), 2019. tug.org/TUGboat/tb40-1/tb124talbot-bib2gls.pdf
- [7] N. Talbot. Logical glossary divisions (`type` vs `group` vs `parent`), 2020. dickimaw-books.com/gallery/?label=logicaldivisions.
- [8] N. Talbot. Sorting, 2019. dickimaw-books.com/gallery/?label=bib2gls-sorting.

◊ Nicola L. C. Talbot
School of Computing Sciences
University of East Anglia
Norwich NR4 7TJ
United Kingdom
<https://www.dickimaw-books.com>

Glossary

cauliflower type of **cabbage** with edible white flower head 1
courgette immature fruit of a **marrow**

eggplant *see aubergine*

marrow long white-fleshed gourd with green skin 1, *see also courgette*

zucchini *see courgette*

Figure 4: `makeindex` can't detect dependent entries that haven't been indexed

Glossary

cabbage vegetable with thick green or purple leaves 1
cauliflower type of **cabbage** with edible white flower head 1
courgette immature fruit of a **marrow**

eggplant *see aubergine*

marrow long white-fleshed gourd with green skin 1, *see also courgette*

zucchini *see courgette*

Figure 5: A second run is required when `\gls` is used in the description

Glossary

cabbage vegetable with thick green or purple leaves
cauliflower type of **cabbage** with edible white flower head 1
courgette immature fruit of a **marrow**

marrow long white-fleshed gourd with green skin 1, *see also courgette*

Figure 6: `bib2gls` with `selection=recorded` and `deps`

Glossary

cabbage vegetable with thick green or purple leaves
cauliflower type of **cabbage** with edible white flower head 1
courgette immature fruit of a **marrow**

marrow long white-fleshed gourd with green skin 1, *see also courgette*

zucchini *see courgette*

Figure 7: `bib2gls` with `selection=recorded` and `deps` and `see`

Making Markdown into a microwave meal

Vít Novotný

Abstract

In today's academic publishing, many venues request L^AT_EX source in addition to or instead of PDF documents. This is often for the purpose of editing and improving full-text search. Services such as arXiv, Editorial Manager, and EasyChair require L^AT_EX source code that can be microwaved in a single run of pdft_EX without shell access and without invoking external programs. This requires that authors include auxiliary files and limits their selection of L^AT_EX packages. In this article, I will show how L^AT_EX documents using the Markdown, Minted, and BIBL^AT_EX packages can be precooked and frozen to be later microwaved in a single run of pdft_EX.

1 Introduction

Academic publishers often require that L^AT_EX documents can be microwaved in a single run of pdft_EX without shell access and without invoking external programs. Precooking and freezing the documents to this form can be a daunting task for authors, who may opt out of using powerful L^AT_EX packages just to save themselves the hassle.

The Markdown package [3, 4, 5] allows the authors to mix the familiar lightweight markup of Markdown with L^AT_EX, but requires shell access and invokes Lua. The Minted package [7] provides a simple interface for typesetting listings with syntax highlighting, but also requires shell access and invokes Python. The BIBL^AT_EX package [2] automates many aspects of bibliography management that require careful manual work in L^AT_EX, such as sorting and formatting, but requires the invocation of Perl.

In this article, I will show by example how a L^AT_EX document using the Markdown, Minted, and BIBL^AT_EX packages can be precooked and frozen, so that it can be later microwaved in a single run of pdft_EX. This will allow the authors to quickly prepare documents using powerful packages without spending a thought on the publisher's demands.

2 Technical details

Here I describe the technicalities of precooking and freezing the outputs of the Markdown, Minted, and BIBL^AT_EX packages before moving on to the example. The famished may skip to the following page.

2.1 Markdown

The Markdown package extracts markdown documents from the main L^AT_EX document to disk and passes control to the Lunamark Lua parser. Lunamark converts the markdown documents to L^AT_EX, saves the converted L^AT_EX documents to disk, and then passes control back to the Markdown package. The Markdown package then inputs the L^AT_EX documents into the main L^AT_EX document for typesetting.

Since version 2.9.0 (2020/09/14), the Markdown package supports the `finalizecache` option that makes Lunamark produce a L^AT_EX document `frozen-Cache.tex` (the frozen cache) mapping an enumeration of markdown documents to their converted L^AT_EX documents, and the `frozencache` option that makes the Markdown package use the frozen cache instead of Lunamark. The resulting L^AT_EX document becomes portable, but further changes in the order and content of markdown documents are not reflected. Appearance of markdown documents can still be adjusted by redefining the L^AT_EX macros that render markdown elements.

Since version 2.2 (2016/06/08), the Minted package supports the `finalizecache` option that makes Minted produce L^AT_EX documents `listing<listing number>.pygtx` (the frozen cache) containing the converted L^AT_EX documents, and the `frozencache` option that makes the package use the frozen cache instead of Pygments. The resulting L^AT_EX document becomes portable, but further changes in the appearance, order, and content of listings are not reflected.

2.2 Minted

The Minted package extracts code listings from the main L^AT_EX document to disk and passes control to the Pygments Python package. Pygments converts the code listings to L^AT_EX, saves the converted L^AT_EX documents to disk, and then passes control back to the Minted package. The Minted package then inputs the L^AT_EX documents into the main L^AT_EX document for typesetting.

Since version 2.2 (2016/06/08), the Minted package supports the `finalizecache` option that makes Minted produce L^AT_EX documents `listing<listing number>.pygtx` (the frozen cache) containing the converted L^AT_EX documents, and the `frozencache` option that makes the package use the frozen cache instead of Pygments. The resulting L^AT_EX document becomes portable, but further changes in the appearance, order, and content of listings are not reflected.

2.3 BIBL^AT_EX

The BIBL^AT_EX package extracts citations from the main L^AT_EX document `<document>.tex` to disk. The author then invokes the Biber Perl package, either manually or using automation software such as GNU Make, L^AT_EXmk, Arara [1], etc. Biber converts the citations and an external bibliography database into a precooked bibliography file `<document>.bb1` and saves it to disk. The BIBL^AT_EX package then inputs the precooked bibliography file for typesetting.

By including the precooked bibliography file with the L^AT_EX source, we remove the necessity of invoking Biber. The resulting L^AT_EX document becomes more portable, but further changes in the order of citations and the appearance and content of references are not reflected.

3 Example

We have the following L^AT_EX document `document.tex` using Markdown, Minted, and BIBL^AT_EX:

```
\documentclass{article}
\usepackage[citations,fencedCode]{markdown}
\markdownSetup{renderers={inputFencedCode=%
  {\inputminted{\#2}{\#1}\write18{\#2 \#1}}}}
\usepackage{minted,biblatex}
\addbibresource{bibliography.bib}
\begin{document}
\begin{markdown*}[hybrid,underscores=false]
The following code in *Python* shows how to
produce iid r.v.'s  $W_1, W_2, \dots, W_k$ 
s.t.  $\prod_i W_i \sim N(0, 1)$ : [Pinelis]
``` python
def pinelis(k, size, iterations=100):
 import numpy as np, numpy.random as npr
 arange1 = np.arange(iterations) + 1
 def eps():
 return (
 npr.uniform(size=size) < 0.5) *
 2.0 - 1.0
)
 def gamma(size):
 return npr.gamma(1.0 / k, size=size)
 def rv():
 return eps() * np.exp(
 np.log(2.0) / (2.0 * k) -
 gamma(size) -
 np.sum(
 gamma((size, iterations)) /
 (2.0 * arange1 + 1.0),
 axis=-1,
) + np.sum(
 np.log(1.0 + 1.0 / arange1) /
 (2.0 * k)
)
)
)
 return [rv() for _ in range(k)]
import matplotlib.pyplot as plt
fig, A = plt.subplots(1, 3)
W1, W2, W3 = pinelis(k=3, size=10**7)
W = (W1, W1 * W2, W1 * W2 * W3)
D = ('W_1', 'W_1W_2', '$W_1W_2W_3$')
for ax, w, desc in zip(A, W, D):
 ax.hist(w, 'auto', density=True)
 ax.set_title(desc)
 ax.set(xlabel='value', ylabel='pdf')
plt.savefig('plot.pdf', dpi=300)
\end{markdown*}
\includegraphics[width=\textwidth]{plot}
\printbibliography
\end{document}
```

In addition to the L<sup>A</sup>T<sub>E</sub>X document, we also have the following bibliography database `bibliography.bib`:

```
@article{pinelis,
 author = {Pinelis, Iosif},
 title = {The exp-normal distribution is
 infinitely divisible},
 journal = {arXiv},
 year = {2018},
}
```

Our L<sup>A</sup>T<sub>E</sub>X document may seem simple, but it requires shell access and invokes Lua, Python, and Perl. A publisher is likely to reject it.

First, we remove the directories `_markdown_document` and `_minted-document` if they exist. Second, we change line 1 of our L<sup>A</sup>T<sub>E</sub>X document to `\documentclass[finalizecache]{article}` and invoke Python, pdflat<sub>E</sub>X, and Biber from the shell:

```
$ pip install numpy scipy matplotlib
$ pdflatex -shell-escape document.tex
$ biber document.bcf
```

Third, we change line 1 of our L<sup>A</sup>T<sub>E</sub>X document to `\documentclass[frozencache]{article}` and we remove `\write18{\#2 \#1}` from line 4. Finally, we create a ZIP archive `document.zip` with the files `document.bbl`, `document.tex`, and `plot.pdf` and with the directories `_markdown_document/`, and `_minted-document/`:

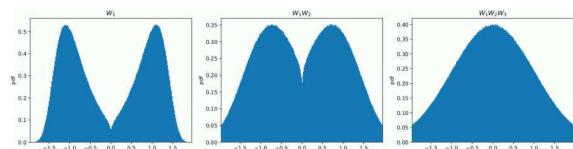
```
$ zip -r document document.{bbl,tex} \
> plot.pdf {_markdown_,_minted-}document/
```

If our document contained forward references, we would also include file `document.aux`. After the publisher has unpacked the ZIP archive, they only need to microwave the document in a single run of pdflat<sub>E</sub>X `document.tex` before savoring it and wolfing it down.

After typesetting, our L<sup>A</sup>T<sub>E</sub>X document produces the following output. Listing was trimmed for brevity (and colors have been grayscaled throughout in the printed article):

The following code in Python shows how to produce iid r.v.'s  $W_1, W_2, \dots, W_k$  s.t.  $\prod_i W_i \sim N(0, 1)$ : [1]

```
def pinelis(k, size, iterations=100):
 import numpy as np
 [...]
 plt.savefig('plot.pdf', dpi=300)
```



### References

- [1] Iosif Pinelis. "The exp-normal distribution is infinitely divisible". (2018).

## 4 Conclusion

Powerful new L<sup>A</sup>T<sub>E</sub>X packages are created every day. However, authors in academia often avoid them to steer clear of the publisher's fury. In this article, I have shown how authors can have their cake and eat it too: by precooking and freezing, the authors can quickly prepare mouth-watering documents using the Markdown, Minted, and BIBI<sup>A</sup>T<sub>E</sub>X packages without having to worry about the publisher's demands.

## Acknowledgements

This work was funded by the South Moravian Centre for International Mobility and the Brno Ph.D. Talent project.

## References

- [1] P. R. M. Cereda. The bird and the lion: arara. *TUGboat* 36(1):37–40, 2015. [tug.org/TUGboat/tb36-1/tb112cereda.pdf](http://tug.org/TUGboat/tb36-1/tb112cereda.pdf)
- [2] P. Kime, P. Lehman. BIBI<sup>A</sup>T<sub>E</sub>X: Sophisticated bibliographies in L<sup>A</sup>T<sub>E</sub>X, 2020. [ctan.org/pkg/biblalex](http://ctan.org/pkg/biblalex)
- [3] V. Novotný. Using markdown inside T<sub>E</sub>X documents. *TUGboat* 38(2):214–217, 2017. [tug.org/TUGboat/tb38-2/tb119novotny.pdf](http://tug.org/TUGboat/tb38-2/tb119novotny.pdf)
- [4] V. Novotný. Markdown 2.7.0: Towards lightweight markup in T<sub>E</sub>X. *TUGboat* 40(1):25–27, 2019. [tug.org/TUGboat/tb40-1/tb124novotny-markdown.pdf](http://tug.org/TUGboat/tb40-1/tb124novotny-markdown.pdf)
- [5] V. Novotný. Markdown: A package for converting and rendering markdown documents inside T<sub>E</sub>X, 2020. [ctan.org/pkg/markdown](http://ctan.org/pkg/markdown)
- [6] I. Pinelis. The exp-normal distribution is infinitely divisible. *arXiv*, 2018. [arxiv.org/abs/1803.09838](http://arxiv.org/abs/1803.09838)
- [7] G. Poore. Minted: Highlighted source code for L<sup>A</sup>T<sub>E</sub>X, 2017. [ctan.org/pkg/minted](http://ctan.org/pkg/minted)

◊ Vít Novotný  
 Nad Cihelnou 602  
 Velešín, 382 32  
 Czech Republic  
 witiko (at) mail dot muni dot cz  
[github.com/witiko](https://github.com/witiko)




---

## User-defined Type 3 fonts in Lua<sup>A</sup>T<sub>E</sub>X

Hans Hagen

## 1 Introduction

This article describe the generic mechanism that is present in Lua<sup>A</sup>T<sub>E</sub>X 1.13 and following to deal with user-defined Type 3 fonts. The examples shown here might not work out well in Con<sup>T</sup>EXt because it has its own font layer, which could interfere with low level hooks, but the same principles apply. Beware: in Con<sup>T</sup>EXt LMTX we do things a bit differently.

In a T<sub>E</sub>X environment Type 3 fonts are normally used for bitmap (pk) fonts. However, they can be useful for other purposes too. In Lua<sup>A</sup>T<sub>E</sub>X a relatively simple mechanism is provided to (ab)use this font format.

## 2 Creation via callback

Defining a whole font in advance when only a few shapes are used makes no sense. Apart from a waste of time and memory it could, as a side effect, trigger the inclusion of all kinds of resources. Therefore, handling is delayed to the moment that the subset of the font actually gets written to the PDF file.

In the frontend you can create virtual characters but their rendering gets in-lined which is often okay, but when you need for instance graphics (using the `image` virtual command) that can be sub-optimal. One can refer to characters in another font and that font can be a (future) Type 3 font. It is only when the document is finalized that the exact subset of glyphs used in the font is known so that is the moment when we deal with what needs to be included. This is done with a plug-in: a single callback that does several things in sequence.

The glyphs in a Type 3 font are streams of PDF operators, a.k.a. char procs. When these are (inline) bitmaps or graphic operators all is relatively easy, but what if they are images or references to shapes from fonts? In that case we also need to make sure that resources are dealt with. We can cook up a complex system of additional resource management, comparable to pages and reused boxes, but it doesn't pay off. Instead we provide a couple of calls to the same callback, `provide_charproc_data`, to deal with that. Because we can use T<sub>E</sub>X asynchronously (using the mechanism for executing tokens) the relevant renderings can be done on demand.

When a Type 3 font is specified and when its `psname` property is equal to the string `none`, a callback is triggered. Actually it is triggered three times.

- The first call is a preroll. It can be used to do the preparations needed for successive calls.

Between the first two calls the used characters of fonts are identified again. This makes it possible to use a reference to an xform in the mentioned char proc stream that itself uses fonts, or we can refer to other fonts directly. As so-called xforms objects are managed independently they don't interfere with the font at hand. The first argument is 1 which indicates that a preroll is being done. The callback function also gets the font id and character reference passed and no return value is expected.

- The second call gets passed the number 2, the font id, and the character index but this time there have to be two return values: the width (in basepoints) and an object number of the char proc stream object. When an object number is returned, a reference will be added to the resource dictionary of the font.
- The third and last call is for housekeeping. This call gets the number 3 passed and the font id. The two expected return values are the scale factor in the font matrix (e.g. 0.001) and a string that has additional entries in the resource dictionary.<sup>1</sup>

Mechanisms like this are normally kept hidden from the user. An example follows in a moment, but first we explain the steps. For sure one needs more code to integrate it properly. Don't do it this way in ConTeXt and expect it to work forever, because we wrap and overload. Anyway, in the end there are only a few cases to cover:

- A stream of mere graphical operators with no dependencies on resources like fonts or objects.
- A stream with a reference to an xform which has the actual content, in which case we need to add a reference in the xobject resource dictionary of the font's.
- A stream with a reference to a font, in which case we need to add a reference font resource dictionary of the font's.
- A stream of operators that do have dependencies on whatever resources one can think of, in which case we need to be able to add these to the fonts resource dictionary.

And, because we can have additional fonts used (either in a created xform or in the stream) we need to analyze the Type 3 fonts first. We assume that

---

<sup>1</sup> An earlier version had four separate calls: one for the scale, two that looped (by multiple calls) over lists of xobjects and used fonts, and a final one for additional resources. But because this mechanism is not meant for general use, assembling the right entries is now delegated to the caller.

no nested Type 3 fonts are used. We also assume that we handle all this at the Lua end.

This mechanism is pretty low level, for a good reason: we're already wrapping up the PDF file so we cannot burden the engine too much with arbitrary actions that mess up the process. Now, one can use TeX to typeset the stream but in practice the stream can best be constructed manually. One can always use TeX to construct an xobject that gets referred to. The good thing is that this feature doesn't change (or add) anything to the front-end.

We could have stuck to a more automated mechanism, for instance by expecting xform object reference, a width, height and depth (indicating some shift) but then we also need to pass information about using d0 or d1 so in the end one needs to know about charprocs anyway and then we can as well expect stream objects. A bit of a complicated mess is compensated for by flexibility, but a mess it remains. In a similar fashion using one callback with numbers indicating each call's purpose is nicer than three different callbacks.

### 3 Examples

It is now time for a few examples. These are simple ones, as it makes no sense to come up with many pages of how to do this in for instance ConTeXt (MkIV that is). We define a font with several solutions mixed. It is not part of some font system. The following example will work okay in MkIV (because we typeset the LuaTeX manual with it).

First we define a couple of token registers and fill them with some content which as you can see can be anything.

```
\newtoks \MoreCrapA
\newtoks \MoreCrapB
\newcount\MoreCrapC

\MoreCrapA{\setbox0\hbox{%
 \font\foo=dejavusansmono at 10bp\foo xyz}}
\MoreCrapB{\setbox0\hbox{%
 \externalfigure[cow.pdf] [height=4mm]}}
```

We also define a simple handler mechanism but hook into the ConTeXt one if we run that macro package (this hook is there only for the manual).

```
\startluacode
 if context then
 RegisterTypeThreeHandler
 = fonts.handlers.typethree.register
 else
 local typethree = { }
 callback.register("provide_charproc_data",
 function(action,f,...)
 if typethree[f] then
 return typethree[f](action,f,...)
```

```

 end
 end)
 function RegisterTypeThreeHandler(id,
 handler)
 typethree[id] = handler
 end
end
\stopluacode

Next we hard code a font table. Later we will see what these character definitions do. Setting psname to none signals that we want to trigger the callback.

\startluacode
local d = 655360

local f = {
 -- the minimal amount of metadata:

 ["name"] = "MyFancyTestFont",
 ["psname"] = "none", -- trigger
 ["format"] = "type3",
 ["tounicode"] = true,

 -- the minimal number of parameters:

 ["parameters"] = {
 ["extra_space"] = 0,
 ["quad"] = d,
 ["size"] = d,
 ["slant"] = 0,
 ["space"] = d/2,
 ["space_shrink"] = d/10,
 ["space_stretch"] = d/6,
 ["x_height"] = d/2,
 },
 -- five characters:

 ["characters"] = {
 [100] = {
 ["commands"] = {
 { "down", d/3 },
 { "rule", d, d },
 },
 ["depth"] = d/3,
 ["height"] = 2*d/3,
 ["width"] = d,
 ["tounicode"] = "0064",
 },
 [101]={
 ["depth"] = 0,
 ["height"] = d/2,
 ["width"] = d,
 ["tounicode"] = "0065",
 },
 [102]={
 ["depth"] = d/3,
 ["height"] = 2*d/3,
 ["width"] = d,
 }
 }
}

```

```

 ["tounicode"] = "0066",
},
[103]={
 ["depth"] = d/4,
 ["height"] = d/2,
 ["width"] = d,
 ["tounicode"] = "0067",
},
[104]={
 ["depth"] = 0,
 ["height"] = d/2,
 ["width"] = d,
 ["tounicode"] = "0068",
},
}

-- normally you do this at the TeX end and
-- integrate into a font definition mechanism
id = font.define(f)

token.set_macro(
 "MyTestFont",
 "\setfontid " .. tostring(id) .. "\relax "
)
tex.setcount(
 "MoreCrapC",
 id
)
\stopluacode

The font is defined and as you can see, we don't need to have any meaningful rendering yet; that is what we do next. Now, if you don't get what happens here by looking at it, this mechanism is not for you. We're talking rather low-level PDF combined with the interface to PDF objects and streams.

For this example font the preroll step will construct some boxes with content. The flushed objects are later referenced by a name (/Xnnn in our case) bound to a form object (m 0 R). Before the assembly stage kicks in, the backend will check what fonts are used again so that referenced fonts get included. The assemble routine uses low level Type 3 directives that are explained in the PDF reference manuals. You have to make sure that no tricky dependencies on other Type 3 fonts occur. The wrapup function takes care of communicating the used resources.

\startluacode
local usedobjects = { }
local usedfonts = { }
local usedfontid = tex.getcount("MoreCrapC")

local function preroll(f,c)
 if c == 103 then
 tex.runtoks("MoreCrapA")

```

```

usedobjects[c]
 = tex.saveboxresource(0,nil,nil,true)
elseif c == 104 then
 tex.runtoks("MoreCrapB")
 usedobjects[c]
 = tex.saveboxresource(0,nil,nil,true)
end
end

local function assemble(f,c)
 if c == 101 then
 local r = pdf.immediateobj(
 "stream",
 "1000 0 d0 10 w 0 1 0 rg "
 .. "0 0 1000 500 re F"
)
 return r, 10
 elseif c == 102 then
 local r = pdf.immediateobj(
 "stream",
 "1000 0 d0 10 w 1 0 0 rg "
 .. "0 -333 1000 1000 re F"
)
 return r, 10
 elseif c == 103 then
 local r = pdf.immediateobj(
 "stream",
 "1000 0 d0 55 0 0 100 0 -200 cm /X103 Do"
)
 return r, 10
 elseif c == 104 then
 local r = pdf.immediateobj(
 "stream",
 "1000 0 d0 55 0 0 50 60 -50 cm /X104 Do"
)
 return r, 10
 else
 return 0, 0
 end
end

local function wrapup(f,c)
 local resources = ""

 if next(usedobjects) then
 local t = { }
 for k, v in pairs(usedobjects) do
 table.insert(t,"/X" .. k .. " " .. v
 .. " 0 R ")
 end
 resources = resources .. "/XObject << "
 .. table.concat(t) .. ">>"
 end

 if next(usedfonts) then
 local t = { }
 for k, v in pairs(usedfonts) do
 table.insert(t,"/F" .. k .. " " .. v
 .. " 0 R ")
 end
 end
end

end

resources = resources .. "/Font << "
.. table.concat(t) .. ">>"
end

return 0.001, resources
end

local function usedfonthandler(action,...)
 if action == 1 then
 return preroll(...)
 elseif action == 2 then
 return assemble(...)
 elseif action == 3 then
 return wrapup(...)
 else
 -- won't happen
 end
end

RegisterTypeThreeHandler(usedfontid,
 usedfonthandler)
\stopluacode

```

The last thing we do is register this plug-in. An example of using the font is this:

```
\MyTestFont
\char100\char101
\char100\char102
\char100\char103
\char100\char104
\char100
```

And the output (grayscale for print; the second character is a green bar and the fourth is a red square):



So, to summarize what we do: the implemented method lives in the backend and leaves the frontend untouched. The backend recognizes a user Type 3 font, and just injects references to charproc streams that can, but are not required to, refer to one xform per charproc. This is about as simple as it could be made with only minimal overhead but it (probably) still has enough potential.

As with the rest of those Lua driven features of LuaTeX, you don't need to be a mastermind to cook up solutions. Of course you should limit yourself to what really makes sense. That said, practice has shown that often whatever opening the program provides, it will be abused, and I expect the same for this mechanism. Just don't blame the engine when the produced PDF misbehaves.

◇ Hans Hagen  
<http://pragma-ade.com>

## Data display, plots and graphs

Peter Wilson

### 1 Introduction

Some years ago [tex.stackexchange.com](https://tex.stackexchange.com) (TeX.SE) seems to have taken over from [comp.text.tex](https://comp.text.tex) for asking about (L<sup>A</sup>)TeX and friends. A perennial question on TeX.SE seems to be asking what (L<sup>A</sup>)TeX is useful for apart from typesetting mathematical papers. There have been many answers to this and I would like to suggest one more: displaying data. In this note I'll mention a couple of ways that I found that L<sup>A</sup>TeX could help with tables, graphs, and plots of data. The impetus for this was when I was strongly advised by my local hospital to keep a check on my blood pressure (BP).

### 2 Practicalities

Following the consultant's suggestion I measure my BP three times a day (morning, afternoon, and in the evening) and average them to get a reading for the day. I do this every day and it is surprising, to me at least, how it varies. I felt that I needed to keep a record of all this so I could present it to the medical experts in case of any problems (like blackouts or falling downstairs—don't ask).

I decided that I needed at least three kinds of records: a tabulation of the BP readings; a plot of the BP; and a graph of the BP.

In the following the data shown is for a hypothetical individual I have designated as Q<sup>1</sup> and have no relationship with any actual BP readings.

### 3 Tabulation

I just used the normal `table` environment with the `booktabs` package to produce a tabulation along the following lines, resulting in the example below for Q at a single day per week (Wk.).

```
% \usepackage{booktabs} % in the preamble
\begin{tabular}{lcllll} \toprule
Date & Wk. & Morn. & Aft. & Eve. & Average \\
& & & \\ \midrule
4/4 & 1 & & & 179/109 & \\
& & & & 179/109 & \\
11/4 & 2 & 156/109 & 147/89 & 149/93 & \\
& & & & 150/97 & \\
etc. & & & & & \\ \bottomrule
\end{tabular}
```

<sup>1</sup> I'm a fan of the original Bond books.

	Date	Wk.	Morn.	Aft.	Eve.	Average
	4/4	1			179/109	179/109
	11/4	2	156/109	147/89	149/93	150/97
	18/4	3	158/108	142/92	146/92	149/97
	etc.					

The higher readings are for the systolic (maximum) blood pressure and the lower ones for the diastolic (minimum) pressure during the heartbeat's cycle.

### 4 Plotting

According to the user manual for my BP monitor, the World Health Organization (WHO) have developed a BP classification scheme. I decided that it might be useful to plot the BP against this scheme as shown for the Q individual.

WHO describe 6 regions in their classification. These are: Optimal BP, Normal BP, Normal Systolic, Mild Hypertension, Moderate Hypertension, and Severe Hypertension.

I have used the standard `picture` environment for producing the plot. The only special macros that I used were

```
% bored with typing \makebox(0,0)
\newcommand{\zbox}[1]{\makebox(0,0){#1}}
% plot symbol
\newcommand*\mk{\zbox{\bullet}}
% \plotit{location}{week}
\newcommand{\plotit}[2]{\put(#1){\mk}}
```

The first two to minimise typing and the last for plotting a BP reading at the `\put` location. With `\makebox(0,0){text}` the reference point for plotting 'text' is at the center, vertically and horizontally, of `text`.

This is an outline of the code I used for the picture.

```
\setlength{\unitlength}{0.8cm}
\begin{picture}(8,11)

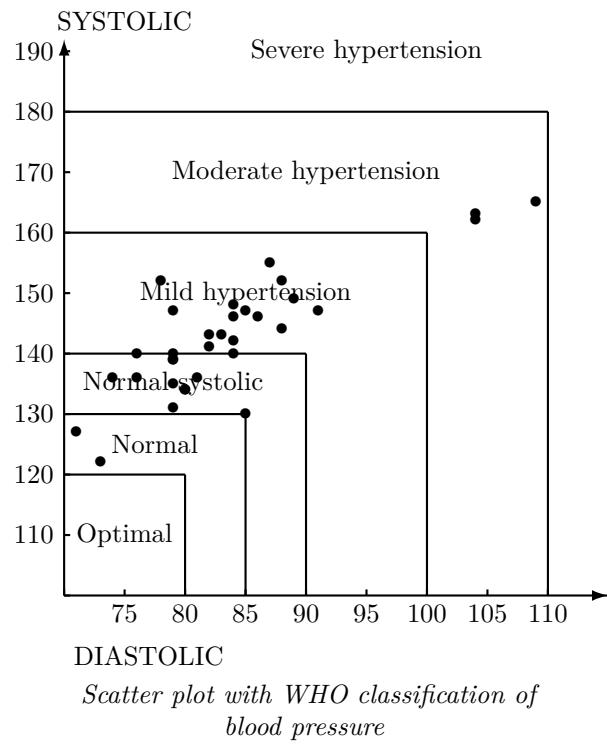
\thicklines

% the horizontal and vertical lines
\put(0,0){\line(1,0){9}}
\put(9,0){\vector(1,0){0}}
\put(0,0){\line(0,1){10}}
\put(0,10){\vector(0,1){0}}
\multimaput(0,0)(1,0){9}{\line(0,1){0.1}}
\multimaput(0,0)(0,1){10}{\line(1,0){0.1}}

% the axis labels
\put(1,10.3){\zbox{SYSTOLIC}}
\put(1.4,-1.0){\zbox{DIASTOLIC}}
\put(1,-0.3){\zbox{75}}
```

```
% etc
\put(8,-0.3){\zbox{110}}
\put(-0.5,1){\zbox{110}}
% etc
\put(-0.5,9){\zbox{190}}
% the regions
\put(0,2){\line(1,0){2}}
\put(2,0){\line(0,1){2}}
\put(1,1){\zbox{0optimal}}
% etc
\put(0,8){\line(1,0){8}}
\put(8,0){\line(0,1){8}}
\put(4,7){\zbox{Moderate hypertension}}
\put(5,9){\zbox{Severe hypertension}}
% the BPs
\plotit[7.8,6.5]{1}
% etc
\plotit{0.8,3.6}{33}
\end{picture}

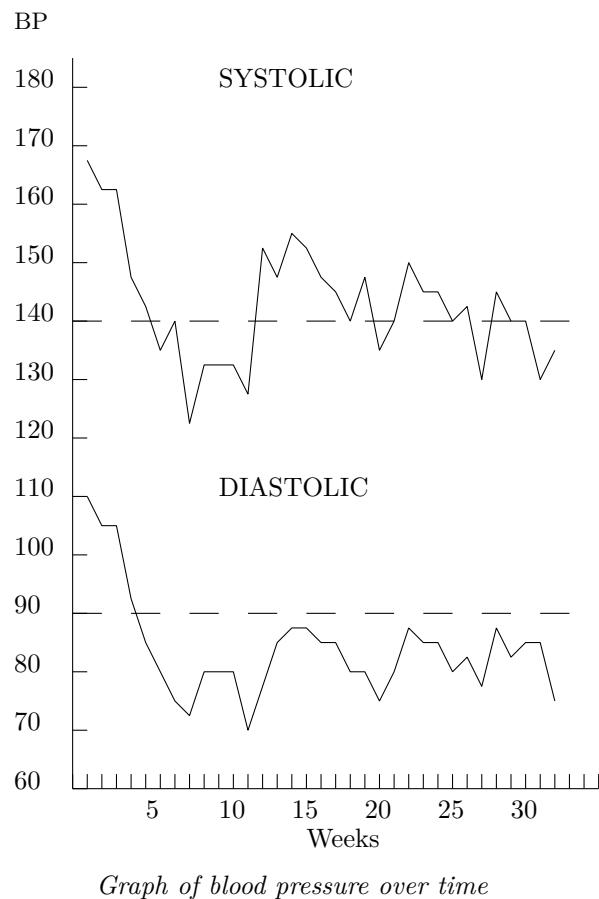
\vspace{10mm}
% caption
{\centering
\emph{Scatter plot with
WHO classification of blood^pressure}
\vspace{\baselineskip}
\par}
```



The result shows that the hypothetical Q person's BP is typically in the range of Normal Systolic to Mild Hypertension but with some outliers.<sup>2</sup>

## 5 Graphing

For graphing BP I used the regular `picture` environment. Nothing special about drawing the axes. The thing of interest here is the use of the `\polyline` macro from the `curve2e` package. This takes a list of coordinates like `(x,y)` and draws straight lines between them.



Here is a brief outline of the code I used for the graph showing the use of `\polyline`.

```
\begin{center}
\setlength{\unitlength}{5.5pt}
\begin{picture}(41,81)
% draw axes, etc., then the BP graphs
% scaled to the size of the axes
% first the systolic
\polyline
```

<sup>2</sup> As a non-medical person I cannot comment on what this might mean for our imaginary person.

```
(6,65)(7,46)(8,46)(9,40)(10,38)(11,35)%
(12,37)(13,30)(14,34)(15,34)(16,34)(17,32)%
(18,42)(19,40)(20,43)(21,42)% etc
% then the diastolic
\polyline
(6,26)(7,23)(8,23)(9,18)(10,15)(11,13)%
(12,11)(13,10)(14,13)(15,13)(16,13)(17,9)%
(18,12)(19,15)(20,16)(21,16)% etc

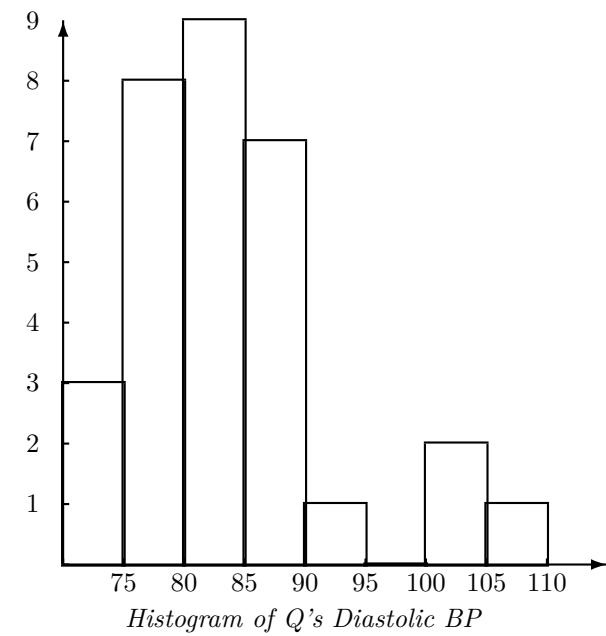
\end{picture}
% caption
\emph{Graph of blood pressure over time}
\vspace{\baselineskip}
\end{center}
```

The dashed lines indicate the upper limits of the WHO Normal Systolic regime.

The graphs show that after an initial worrying period Q's BP settled down to a fairly regular pattern albeit with some fits and starts.

## 6 Histogram

Another way of displaying data is by a histogram which shows the number of data points noted within sets of ranges. The following is a histogram of Q's diastolic BP for 5 mg ranges.



Nothing special about the code. I used the `\framebox` macro for drawing the rectangular regions and created a macro to reduce the number of characters needed for specifying its location and size.

```
\newcommand{\histit}[2]{\put(#1,0.0)%
{\framebox(1,#2){}}}
```

where the first argument is the x location of the framebox and the second is its height.

I must say that I found the scatter plot more informative than the histogram, although the latter highlighted the unusual high diastolic readings.

## 7 Summary

I have shown four different ways of displaying data. Edward Tufte<sup>3</sup> has shown many other ways.

There are many applications for (L)A<sup>T</sup>E<sub>X</sub> and friends. Among those noted on TeX.SE, apart from mathematical and scientific publications, are:

**Books** fiction and non-fiction

**Correspondence**

**Games** Bridge, Chess, Crosswords, Noughts and Crosses (aka Tic-tac-toe), Sudoku

**Greeting cards**

**Invoices**

**Literature** Critical editions, Multilingual

**Mars Rover** (programmed via T<sub>E</sub>X)

**Music**

**Newsletters**

**Poetry**

**Postcards**

**Presentations** (slides)

...

I hope that my small application might give thoughts towards suitable additions to the above list.

◊ Peter Wilson  
12 Sovereign Close  
Kenilworth, CV8 1SQ  
UK  
herries dot press (at)  
earthlink dot net

---

<sup>3</sup> Edward R. Tufte, *The Visual Display of Quantitative Information*, Graphics Press, 1983.

## Short report on the state of LuaTeX, 2020

Luigi Scarso

### Abstract

A short report on the current status of LuaTeX and its relatives: LuaHBTeX, LuaJITTeX and LuaJITHBTeX.

### 1 Background

First, let's summarize that there are four programs or “flavors” of LuaTeX:

- LuaTeX, with `lua`;
- LuaJITTeX, with `luajit` (just-in-time compilation capability);
- LuaHBTeX, with `lua` and HarfBuzz;
- LuaJITHBTeX, with `luajit` and HarfBuzz.

The build system manages the task of compiling and linking the shared components and the common components, so that, for example, they all have exactly the same TeX core, and share the same version number and development id.

On 30 May 2019 the first commit of LuaHBTeX, a LuaTeX variant with the ability to use HarfBuzz for glyph shaping, landed in the LuaTeX repository, after a discussion started around mid-February about the merging of HarfTeX by Khaled Hosny and the upcoming LuaTeX in TeX Live 2019. By that time LuaTeX was already frozen for the DVD and, materially, it was not possible to reopen the development. Also in 2019, LuaTeX entered its “bug fixing” phase (more on this below), further complicating the merge.

LuaHBTeX was integrated in the experimental branch of LuaTeX repository on 6 July 2019 and directly after, on 8 July 2019, it landed in the TeX Live repository; The first release of LuaHBTeX, tagged as version 1.11.1, was on 19 October 2019, giving a wide margin for testing for the next (i.e., now the current) TeX Live 2020.

Together with HarfBuzz, the other component under “observation” in 2019 was the `pplib` library, the PDF reader library by Paweł Jackowski. This was the candidate to replace `poppler` in TeX Live—this eventually happened with the first commit to the TeX Live repository on 21 April 2020. This means that the next TeX Live 2021 will entirely use `pplib` instead of `poppler`, for both LuaTeX (as was the case in previous years), and now also XeTeX; `poppler` is no longer in the TeX Live repository. (By the way, pdftex will continue to use its own semi-homegrown `libxpdf` to read PDF files until there is some clear reason to change.)

MetaPost gets related special treatment: the library in LuaTeX includes only the decimal and

the floating-point mode, while the `mpost` program also includes the `mpfr` library for arbitrary precision support.

As result of mixing and matching all these variations, building LuaTeX and integrating it into TeX Live is quite a complex task, but thanks to the GNU autotools, things are manageable.

### 2 The current status of LuaTeX

As noted above, LuaHBTeX (version 1.12.0) shipped for the first time with the TeX Live 2020 DVD, and it is already supported by LuaTeX: At the TeX Live 2020 meeting, the talk “HarfBuzz in LuaTeX” by Marcel Krüger has shown some differences between the HarfBuzz text shaping and the ConTeXt text shaping; also better memory management for large fonts with respect to LuaTeX, especially for 32-bit platforms.

On the other side, Petr Olšák in 2020 has published OpTeX, “...a LuaTeX format based on Plain TeX macros (by Donald Knuth) and on OPmac macros” (see [petr.olsak.net/optex](http://petr.olsak.net/optex), and article in this issue), also included in TeX Live 2020. It’s not clear if it will eventually support LuaHBTeX.

Finally, also at the TUG 2020 online meeting, Patrick Gundlach in his talk “Speedata Publisher—a different approach to typesetting using Lua” (see [tug.org/TUGboat/41-2/gundlach-speedata.pdf](https://tug.org/TUGboat/41-2/gundlach-speedata.pdf)) has shown an example of a working workflow that uses LuaTeX (and possibly LuaJITTeX) purely by means of the `lua` API—a sort of TeX without \TeX. The Speedata Publisher software has been actively developed for a decade.

In light of these continuing developments, it is therefore appropriate to clarify the meaning of “bug fixing” mode, because it is sometimes associated with the term “frozen”.

LuaTeX is based on the `lua` release 5.3.5, and it will stay on the 5.3 version at least for TeX Live 2021 and TeX Live 2022, possibly switching to the final release 5.3.6 (in release candidate 2 at the date of 2020-07-23) at some future point. The current release of `lua` is 5.4.0, with approximately five years between two versions; it’s good practice to have an year of transition between two different versions, so a rough estimation for the next `lua` transition is six years from now, i.e., around TeX Live 2026.

On the side of the TeX core, the plan is for bug fixing and marginal improvements, for example the `\tracinglostchars ≥ 3` that now raises an error (a new feature added across engines by David Jones), but not new features. From what we have seen previously, stressing LuaTeX in different areas (e.g., only with the `lua` API or with the new HarfBuzz

shaping library) can reveal hidden bugs, but it should be noted that bug fixing is a complex task because the fix must be well harmonized with the rest of the code: For example, some issues with DVI output that need to be checked carefully are still open.

Nevertheless, there are three areas that are still marked as “under active development”: the first is the `ffi` (foreign function interface) library, that in `LuaTeX` is not yet finished and not as functional as its counterpart in `LuajITTEX`. Admittedly it is not a key feature of `LuaTeX` and probably useful only in the context of automated workflows.

The second is the binding with HarfBuzz library, currently given by the `luahrbuzz` module. If necessary the binding can still be expanded and/or modified, preserving as much as possible the current API, because `LuaHBTeX` is in an early phase of adoption.

The third area is the `pplib` library that surely needs more testing.

Finally, the bug fixing phase certainly also involves the MetaPost library.

### 3 The current status of LuajITTEX

`LuajITTEX` is (or in some way is considered) a niche engine. One issue is that while `LuaTeX` is based on Lua 5.3.5, `LuajITTEX` is still based on 5.1 with some partial coverage of 5.2. `LuajITTEX` also has some intrinsic limits, such as the fixed number of nested tables, which has a serious impact on the table serialization. By design, `LuaTeX` makes heavy use of C functions bound via the classic Lua/C API; the just-in-time (JIT) compiler doesn’t play well in this situation, but this is not a serious issue, given that it can be turned off on demand (and indeed it’s off by default). Finally, `LuajIT` doesn’t support all of Lua’s platforms, although the most important ones are available.

On the other hand, the `LuajIT` virtual machine is much faster than Lua and the compilation of an article can have a significant speed-up. For this article, `LuajITHBTeX` is 2.5 times faster than `LuaHBTeX` with exactly the same `LuaTeX` format; although for complex documents the gain is smaller, around 15%–20%.

The lack of a specific format for `LuajIT` does fake the results a bit, but maintaining an additional format in this case is not an easy task: To take advantage of the JIT, where `LuajIT` shines, one has to write specialized Lua code and using the `ffi` module requires rather in-depth knowledge of C to achieve significant results. Currently only `ConTeXtMkIV` has some support for `LuajITTeX`.

Probably `LuajITTeX` and `LuajITHBTeX` are better suited for specialized tasks (e.g. database publishing) or as software as service in cloud, possibly in a containerized environment, but they should also be considered a research tool in digital typesetting.

Currently `LuajIT` in `\TeX` Live is still using the 2.1-beta3 release (from 2017), but it is likely it will sync with the official repository by the end of the year. Although `LuajIT` development is not proceeding at a rapid pace, there have been important updates (e.g., all `LuajIT` 64-bit ports now use 64-bit GC objects by default; and there is support for more platforms). There are some mismatches with Lua (a few functions in Lua that are not available in `LuajIT`, notably the `utf8` module) still to be fixed.

### 4 Conclusion

At the `\TeX` core, `LuaTeX` and `LuaHBTeX` are exactly the same and the choice between one or the other depends only on whether or not one accepts HarfBuzz as a dependency. As `OpTeX` has shown, `LuahBTeX` is not always the necessary choice. In any case, the current state is better described by “bug fixing mode with marginal improvements” rather than “frozen”, with an emphasis on stability. The area marked as “under active development” may change more significantly, but this should have a minimal impact on stability.

`LuajITTeX` and `LuahBTeX` are more or less still out of the mainstream and that gives a wider range for maneuvering; given the high efficiency of the implementation of `LuajIT`, it’s often better to code a module directly in Lua rather than compile and link a C module. Admittedly, it’s a rather specialized topic, but efficiency has its costs.

◊ Luigi Scarso  
 luigi.scarso (at) gmail dot com

## Distinguishing 8-bit characters and Japanese characters in (u)pTeX

Hironori Kitagawa

### Abstract

pTeX (an extension of TeX for Japanese typesetting) uses a legacy encoding as the internal Japanese encoding, while accepting UTF-8 input. This means that pTeX does code conversion in input and output. Also, pTeX (and its Unicode extension upTeX) distinguishes 8-bit character tokens and Japanese character tokens, while this distinction disappears when tokens are processed with `\string` and `\meaning`, or printed to a file or the terminal.

These facts cause several unnatural behaviors with (u)pTeX. For example, pTeX garbles “f” (long s) to “顛” on some occasions. This paper explains these unnatural behaviors, and discusses an experiment in improvement by the author.

### 1 Introduction

Since TeX Live 2018, UTF-8 has been the new default input encoding in LATEX [8]. However, with pLATEX, which is a modified version of LATEX for the pTeX engine, the source

```
%#!platex
\documentclass{minimal}
\begin{document}f\end{document} % long s
```

gives an inconsistent error message [4] (edited to fit *TUGboat*'s narrow columns):

```
! Package inputenc Error: Unicode character
顛 (U+C4CF) not set up for use with LaTeX.
```

Here “顛”, “f” and U+C4CF are all different characters.

The purpose of this paper is to investigate the background of this message and propose patches to resolve this issue. This paper is based on a cancelled talk [6] in TeXConf 2019.<sup>1</sup>

In this paper, the following are assumed:

- All inputs and outputs are encoded in UTF-8.
- pTeX uses EUC-JP as the internal Japanese encoding (see Section 2.1).
- Sources are typeset in plain pTeX (`pTeX`), unless stated otherwise by `%#!`.
- The notation `<AB>` describes a byte `0xab`, or a character token whose code is `0xab`.

### 2 Overview of pTeX

pTeX is an engine extension of TeX82 for Japanese typesetting. It can typeset Japanese documents of

<sup>1</sup> TeXConf 2019 (the annual meeting of Japanese TeX users, [texconf2019.tumblr.com](https://texconf2019.tumblr.com)) was canceled due to a typhoon.

professional quality [9], including Japanese line breaking rules and vertical typesetting.

pTeX and pLATEX were originally developed by the ASCII Corporation<sup>2</sup> [1]. However, pTeX and pLATEX in TeX Live, which are our concern, are community editions. These are currently maintained by the Japanese TeX Development Community.<sup>3</sup> For more detail, please see the English guide for pTeX [3].

pTeX itself does not have ε-TeX features, but there is ε-pTeX [7], which merges pTeX, ε-TeX and additional primitives. Anything discussed about pTeX in this paper (besides this paragraph) also applies to ε-pTeX, so I simply write “pTeX” instead of “pTeX and ε-pTeX”. Note that the pLATEX format in TeX Live is produced by ε-pTeX, because recent versions of LATEX require ε-TeX features.

#### 2.1 Input code conversion by `ptexenc`

Although pTeX in TeX Live accepts UTF-8 inputs, the internal Japanese character set is limited to JIS X 0208 (JIS level 1 and 2 kanjis), which is a legacy character set before Unicode. pTeX uses Shift\_JIS (Windows) or EUC-JP (other) as the internal encoding of JIS X 0208.

In pTeX and related programs, the `ptexenc` library [12] converts an input line to the internal encoding. pTeX's input processor actually reads the converted result by `ptexenc`. A valid UTF-8 sequence which does not represent a JIS X 0208 character — such as `<C5><BF>` (“f”) or `<C3><9F>` (“ß”) — is converted to `^^ab`.

On the other hand, an invalid UTF-8 sequence is converted into `<A2><AF>` (an undefined code point in EUC-JP) sometimes, in TeX Live 2019 or prior. In TeX Live 2020, the sequence is always converted into `^^-notation`.

#### 2.2 Japanese character tokens

pTeX divides character tokens into two groups: ordinary 8-bit character tokens and Japanese character tokens. The former are not different from tokens in 8-bit engines, say, TeX82 and pdfTeX. A `^^-notation` sequence is always treated as an 8-bit character.

A Japanese character token is represented by its character code. In other words, although there is a `\kcatcode` primitive, which is the counterpart of `\catcode`, its information is *not* stored in tokens. Hence, changing `\kcatcode` by users is not recommended.

<sup>2</sup> Currently ASCII DWANGO in DWANGO Co. Ltd.

<sup>3</sup> [texjp.org/](https://texjp.org/). Several GitHub repositories:  
[github.com/texjporg/tex-jp-build](https://github.com/texjporg/tex-jp-build) ((u)pTeX),  
[github.com/texjporg/platex](https://github.com/texjporg/platex) (pLATEX).

### 2.3 An example input

Now we look at an example. Our input line is

`a<C3><9F><E6><BC><A2><C5><BF><C2><A7>` (あ漢§)

First, pTeXenc converts this line into

`a^^c3^^9f<B4><C1>^^c5^^bf<A1><F8>`

which is fed to pTeX's input processor. The final character “§” is included in JIS X 0208.

From the result above, pTeX produces tokens

`a11 <C3>12 <9F>12 漢 <C5>12 <BF>12 §`

where 漢 and § are Japanese character tokens. From this example, we can see that we cannot write “§” directly to output this character in a Latin font (use commands or ^^c2^^a7).

## 3 Stringization in pTeX

### 3.1 Overview

Names of multiletter control sequences, which include control sequences with single Japanese character name, such as \あ, are stringized, that is to say, they are stored into the string pool. Similarly, some primitives, such as \string, \jobname, \meaning and \the (almost always the case), first stringize their intermediate results into the string pool, and then retokenize these intermediate results.

Stringization of pTeX has two crucial points.

- The origin of a byte is lost in stringization. A byte sequence, for example <C5><BF>, in the string pool may be the result of stringization of a Japanese character “顛”, or that of two 8-bit characters <C5> and <BF>.
- In retokenization, a byte sequence which represents a Japanese character in the internal encoding is *always* converted to a Japanese character token. For example, <C5><BF> is always converted to a Japanese token 顛.

These points cause unnatural behavior, namely bytes from 8-bit characters becoming garbled to Japanese character tokens. We look into several examples.

### 3.2 Control sequence name

Let's begin with the following source:

```
\font\Z=ec-lmr10 \Z % T1 encoding
\expandafter\def\csname uf\endcsname{AA}
\expandafter\def\csname u顛\endcsname{BB}
\def\ZZ#1{\#1 (\string#1)}
\expandafter\ZZ\csname u^^c5^^bf\endcsname% (1)
\expandafter\ZZ\csname uf\endcsname % (2)
\expandafter\ZZ\csname u顛\endcsname % (3)
```

With pTeX, (1)–(3) produces the same result

BB (＼u顛)

This is because all of

```
\csname u^^c5^^bf\endcsname
\csname uf\endcsname % f: <C5><BF> in UTF-8
\csname u顛\endcsname % 頛: <C5><BF> in EUC-JP
```

have the same name u<C5><BF> in pTeX, hence they are treated as the same control sequence. Applying \string to them, we get the same token list

`\u12 u12 顛`

This explains the error message in the introduction. “顛 (U+C4CF)” in the message is generated from

```
\expandafter\string
\csname u8:\string<C5>\string<BF>\endcsname
```

The inputenc package expects that applying \string to the above control sequence produces

`\u12 u12 812 :12 <C5>12 <BF>12`

but the result in pLaTeX is

`\u12 u12 812 :12 頛`

### 3.3 \meaning

The result of

```
\font\Z=ec-lmr10 \Z % T1 encoding
\def\fuga{^^c5^^bf顛}\meaning\fuga
```

differs between plain TeX and plain pTeX:

**plain TeX** macro:->Å£éäŽÅ£

**plain pTeX** macro:->顛顛顛

Now we look at what happened with pTeX. The definition of \fuga is represented by the token list

`<C5>12 <BF>12 頛 <C5>12 <BF>12`

This gives the following string as the intermediate result of \meaning.

`macro:-><C5><BF><C5><BF><C5><BF>`

Retokenizing this string gives the final result

`macro:->顛顛顛`

which we have already seen.

### 3.4 A tricky application

The behavior described in Section 3.2 has a tricky application: generating a Japanese character token from its code number, even in an expansion-only context. This can be constructed as follows:

```
%#!epstopdf
\font\Z=ec-lmr10 \Z % T1 encoding
\input expl3-generic % for \char_generate:nn
\ExplSyntaxOn
\cs_generate_variant:Nn \cs_to_str:N { c }
```

```
\cs_new:Npn \tkchar #1 {
\cs_to_str:c {
\char_generate:nn % upper byte
{ \int_div_truncate:nn { #1 } { 256 } }
{ 12 }
\char_generate:nn % lower byte
{ \int_mod:nn { #1 } { 256 } }
{ 12 }
}
\ExplSyntaxOff
\edef\A{\tkchar{\`漢}\tkchar{`字}}
\meaning\A % ==> macro:->漢字
```

This `\tkchar` will be unnecessary as of TeX Live 2020, since the `\Uchar` and `\Ucharcat` primitives were added into ε-pTeX at that time.

## 4 Output to file or terminal

### 4.1 Output code conversion

As with input, pTeX does a code conversion from the internal Japanese encoding to UTF-8 in outputting to a file or the terminal. This is done in two steps:

- As with TeX82, pTeX uses the `print` procedure for printing a string.<sup>4</sup> In pTeX, a byte is printable if and only if its value is between 32 (“\_”) and 126 (“~”), or it is used in the internal Japanese encoding (`<A1>-<FE>` in EUC-JP).
- pTeX uses the `putc2` function instead of the standard `putc` C function. `putc2` is a variation of `putc` with code conversion, and is defined in `ptexenc`.

Hence pTeX may garble 8-bit characters, such as `<C5><BF>`, into a Japanese character in output. We look into two examples, one is of `\write` and the other is of `\message`.

### 4.2 `\write`

With pTeX, the following source

```
\newwrite\OUT
\immediate\openout\OUT=test.dat
\immediate\write\OUT{\顛fB}
\immediate\closeout\OUT
```

produces a file `test.dat`, whose contents are

`顛顛<C3>^^9f`

Let's look at what happened.

First, the argument of `\write` is (expanded to) the following token list.

`顛 <C5>12 <BF>12 <C3>12 <9F>12`

<sup>4</sup> In fact, `slow_print` is used for printing a string which might contain unprintable characters. However, `slow_print` calls `print` internally.

Then, pTeX prints this token list. Since `<A1>-<FE>` are printable and `<9F>` is not, the `putc2` function receives the following string, one byte per call.

`<C5><BF><C5><BF><C3>^^9f`

Each `<C5><BF>` is converted to “顛” by `putc2`, while the single `<C3>` remains unchanged. Hence the final result is “顛顛`<C3>^^9f`”, as shown.

### 4.3 `\message`

`\message` is similar to `\write`, but differs in that it stringizes its argument. Now consider an input line

`\message{^^fe^^f3:顛:}`

Here `顛` (`<F0><AA><9A><B2>` in UTF-8) is a character included in JIS X 0213, but not in JIS X 0208.

The argument of `\message` is (expanded to) the following token list.

`<FE>12 <F3>12 :12 <F0>12 <AA>12 <9A>12 <B2>12 :12`

Then, this token list is stringized to

`<FE><F3>:<F0><AA><9A><B2>:`

This string is “printed” by `print`; since only `<9A>` is unprintable, `putc2` receives

`<FE><F3>:<F0><AA>^^9a<B2>:`

Now, `putc2` converts `<FE><F3>` (an undefined code point in EUC-JP) to the null character `<00>`, and `<F0><AA>` to “險”. Hence the final result is

`<00>:险^^9a<B2>:`

### 4.4 Controlling printability

TeX82 and pdftEX support TCX (TeX Character Translation) files [2], which can be used to specify which characters are printable. In fact, `cp227.tcx` is activated in (pdf)LATEX and several other formats in TeX Live, to make characters 128–255 and three control characters printable. One can switch to a different TCX file at runtime. For example, only characters 32–126 are printable in

`latex -translate-file=empty.tcx`

However, pTeX was not expected to use TCX files (no TCX files are activated in formats by pTeX in default). inipTeX can make characters printable by a TCX file, and that's all. For example, to make characters 128–255 printable in pTeX, one has to make another format with appropriate option. There is no method to make an arbitrary character, say `<AO>`, unprintable when using this format.

## 5 upTeX

### 5.1 Overview

upTeX [10, 11] is a Unicode extension of pTeX by Takuji Tanaka. upTeX is (almost fully) upward-compatible with pTeX, so it is a very convenient solution for converting existing documents to Unicode with minimal changes.

In upTeX, a Japanese character token is a pair of the character code and `\kcatcode`. Furthermore, `\kcatcode` controls whether a UTF-8 sequence produces a Japanese character token or a sequence of 8-bit tokens. For example, `<E9><A1><9B>` (顛, U+985B) in an input line is treated as three 8-bit characters when `\kcatcode"985B` is 15, and as a Japanese character otherwise.

### 5.2 No code conversion

Since upTeX's internal Japanese character code is Unicode (UTF-8 in the string pool), code conversion by `ptexenc` has no effect. Hence the inconsistent error message described in the introduction will not be issued.

### 5.3 Retokenization and `\kcatcode`

In upTeX, `\kcatcode` is involved in the retokenization process. Specifically, a UTF-8 sequence is converted into a Japanese character token if and only if its `\kcatcode` is not 15. This means that the result of `\meaning` of the same macro depends on `\kcatcode` settings, as in the following example.

```
%#!uptex
\font\Z=ec-lmr10 \Z % T1 encoding
%% default: \kcatcode"3042=17
\def\hoge{^e3^81^82あ}
\kcatcode"3042=15
\meaning\hoge % ==> macro:->ääääää
\kcatcode"3042=17
\meaning\hoge % ==> macro:->ああ
```

The definition of `\hoge` is represented by the token list

`<E3>_12 <81>_12 <82>_12 あ_17`

Hence the intermediate result of `\meaning\hoge` is

`macro:-><E3><81><82><E3><81><82>`

However, because the `\kcatcode` of “あ” is changed, two calls of `\meaning\hoge` give different results.

We will see results of `\string` of multiletter control sequences later.

## 6 Distinguishing bytes from 8-bit characters and those from Japanese characters

To resolve (u)pTeX's behavior described so far, I have been developing an experimental version<sup>5</sup> of (u)pTeX, where stringization and outputting retain the origin of a byte—an 8-bit character (token) or a Japanese one. I refer to these as “experimental”, and (u)pTeX in TeX Live development repository as “trunk”.

The implementation approach is to extend the range of a “byte” to 0–511 (Table 1). A value between 0–255 means a byte from an 8-bit character (token), and 256–511 means a “byte” from a Japanese one.

I tested a different approach, namely using `<FF>` as a prefix to a byte 128–255 which came from an 8-bit character. But this approach caused confusion with `<FF>`, so I gave up.

### 6.1 `\write`

For example, consider the source from Section 4.2:

```
\newwrite\OUT
\immediate\openout\OUT=test.dat
\immediate\write\OUT{顛fß}
\immediate\closeout\OUT
```

with the experimental pTeX. When no TCX file is activated, `putc2` receives the string

`<1C5><1BF>^c5^bf^c3^9f`

because a Japanese token 顛 sends `<1C5><1BF>` to `putc2`, and `<80>-<FF>` are not printable. Thus the contents of the output `test.dat` are

`顛^c5^bf^c3^9f`

When `cp227.tcx` is activated, they become

`顛fß`

because `<80>-<FF>` are printable in this case.

### 6.2 The string pool

Since the range of a “byte” is increased to 0–511, the type of the string pool is changed to let each element store a “byte”; concretely, to a 16-bit array. For example, let's reconsider the following source:

```
\font\Z=ec-lmr10 \Z % T1 encoding
\def\fuga{^c5^bf顛f}\meaning\fuga
```

With the experimental pTeX, the intermediate result of `\meaning\fuga` is

`macro:-><C5><BF><1C5><1BF><C5><BF>`

Hence the result of `\meaning\fuga` is

---

<sup>5</sup> [github.com/h-kitagawa/tex-jp-build/tree/printkanji\\_16bit](https://github.com/h-kitagawa/tex-jp-build/tree/printkanji_16bit). GitHub issue: [5]

**Table 1:** A “byte” in experimental (u)pTeX

“byte” $c$	0–255	256–511
origin	an 8-bit character (token)	a Japanese character (token)
printable characters	32–126 (“ $\sqcup$ ”–“ $\sim$ ”)*	all
“safe” printing of $c$	$\text{print}(c)$	$\text{print\_char}(c)$ (not $\text{print}$ )
$\text{putc2}(c, \dots)$	<i>without</i> code conversion	<i>with</i> code conversion**
retokenization	an 8-bit character token $c$	a Japanese character token**

\* Web2C’s default; can be extended by a TCX file.

\*\* With adjacent “bytes” which are between 256–511.

`macro:->\f 順 \f`

because only <1C5><1BF> is converted to a Japanese character token 順.

The change in the type for the string pool increases the size of format files by about the total length of strings, but the amount of increase is not so large. For example, the `platex-dev` format is increased by about 3.5% (see table below). As of TeX Live 2020, pdfTeX and (u)pTeX use compressed format files, so the amount of increase on disk is smaller.

<code>platex-dev.fmt</code> [kB]	trunk	experimental
uncompressed	10412	10774
compressed	2322	2380

I wanted to keep the modification as small and simple as possible; so I left unchanged the structure of the string pool, except for adding a “flag bit”.

### 6.3 Control sequence names in upTeX

In the experimental pTeX,

```
\csname uf\endcsname
\csname u顛\endcsname
```

are treated as different control sequences. This is because the name of the former is u<C5><BF>, while that of the latter is u<1C5><1BF>. This behavior seems to be natural.

However, the situation is more arguable between the experimental upTeX and the trunk upTeX. For example, let’s compare the results of (1) and (2) in the following source by both versions of upTeX.

```
%#!uptex
\font\Z=ec-lmr10 \Z % T1 encoding
\def\ZZ#1{\#1 (\string#1)}
\kcatcode"3042=15
\expandafter\def\csname あ\endcsname{AA}
\kcatcode"3042=17
\expandafter\def\csname あ\endcsname{BB}
\kcatcode"3042=17 \expandafter\ZZ
\csname あ\endcsname % (1)
```

```
\kcatcode"3042=15 \expandafter\ZZ
\csname あ\endcsname % (2)
```

Results are summarized in Table 2. One may feel uneasy about both results.

**trunk** The results of `\string` for (1) and (2) differ, while they represent the same control sequence (as in Section 5.3).

**experimental** (1) and (2) represent different control sequences.

### 6.4 Input buffer(s)

I also introduced an array `buffer2` as a companion array to `buffer`, which contains an input line. `buffer2[i]` plays the role of the “upper byte” of `buffer[i]`. Hence, when (u)pTeX considers a byte sequence `buffer[i .. j]` as a Japanese character, `buffer2[i .. j]` is set to 1. This is needed when scanning a control sequence name in order to distinguish a byte which consists a part of a Japanese character from another byte.

Suppose that the category codes of <C5> and <BF> are both 11 (letter), an input line contains

```
\<C5><BF>^^c5^^bf (\顛^^c5^^bf, \顛f) (1)
```

and pTeX is about to scan this control sequence (1). Since (p)pTeX converts `^^`-notation in a control sequence name into single characters in `buffer`, the contents of `buffer` become

```
\<C5><BF><C5><BF> (\顛顛)
```

Thus, the control sequence (1) cannot be distinguished from `\顛顛` so far. However, the experimental pTeX can distinguish the control sequence (1) from `\顛顛`, because the contents of `buffer2` differ (see Table 3).

`buffer2` is also useful in showing contexts in upTeX. For example, let’s look the following input:

```
%#!uptex
\def\J{\kcatcode"3042=17 }
\def\L{\kcatcode"3042=15 }
\J あ\L あ\undefined あ\J あ
```

**Table 2:** Properties of `\csname` と `\endcsname` of TeX source in Section 6.3

\kcatcode of “あ”	trunk		experimental	
	name	result of \TEST	name	result of \TEST
(1) 17	<u>&lt;E3&gt;&lt;81&gt;&lt;82&gt;</u>	BB (\あ)	<u>&lt;1E3&gt;&lt;181&gt;&lt;182&gt;</u>	BB (\あ)
(2) 15	<u>&lt;E3&gt;&lt;81&gt;&lt;82&gt;</u>	BB (\ãAĆ)	<u>&lt;E3&gt;&lt;81&gt;&lt;82&gt;</u>	AA (\ãAĆ)

**Table 3:** Contents of `buffer` and `buffer2` when the experimental pTeX scans control sequences in an input line

\顛^^c5^^bf (\顛f)					\顛顛						
buffer	\	<u>&lt;C5&gt;</u>	<u>&lt;BF&gt;</u>	<u>&lt;C5&gt;</u>	<u>&lt;BF&gt;</u>	\	<u>&lt;C5&gt;</u>	<u>&lt;BF&gt;</u>	<u>&lt;C5&gt;</u>	<u>&lt;BF&gt;</u>	
buffer2	0	1	1	0	0	0	1	1	1	1	
name		<u>&lt;1C5&gt;&lt;1BF&gt;&lt;C5&gt;&lt;BF&gt;</u>					<u>&lt;1C5&gt;&lt;1BF&gt;&lt;1C5&gt;&lt;1BF&gt;</u>				

With the experimental upTeX (and no TCX file), we can know that the second “あ” is treated as three 8-bit characters from the error message. I hope this will be useful in debugging.

```
! Undefined control sequence.
1.3 \J あ\L ^^e3^^81^^82\undefined
 あ\J あ
```

The third and the final “あ” is not read by upTeX’s input processor at the error. So they are printed as if all UTF-8 characters gave Japanese character tokens.

## 7 Conclusion

The primary factor of the complications discussed in this paper is that (u)pTeX are Japanese extension of an 8-bit engine; this causes the same byte sequence can represent different things, namely a sequence of 8-bit characters (token) or Japanese characters. Although my experiment does not get rid of this factor (only ameliorates it), I hope that it is helpful.

I thank the executive committee of TeXConf 2019, which gave me the opportunity for preparing the original talk, and the people who discussed the topics of this paper with me, especially Hironobu Yamashita, Takuji Tanaka, Takayuki Yato, and Norbert Preining.

## References

- [1] ASCII Corporation. ASCII Japanese TeX (pTeX) (in Japanese). [asciidwango.github.io/ptex/index.html](https://asciidwango.github.io/ptex/index.html).
- [2] K. Berry, O. Weber. Web2c, for version 2019. [tug.org/texlive/Contents/live/texmf-dist/doc/web2c/web2c.pdf](https://tug.org/texlive/Contents/live/texmf-dist/doc/web2c/web2c.pdf), Feb. 2019.
- [3] Japanese TeX Development Community. Guide to pTeX and friends. [ctan.org/pkg/ptex-manual](https://ctan.org/pkg/ptex-manual).

- [4] JulienPalard. Inconsistent error message. [github.com/texjporg/platex/issues/84](https://github.com/texjporg/platex/issues/84).
- [5] H. Kitagawa. Distinction between a byte sequence and a Japanese character token (in Japanese). [github.com/texjporg/tex-jp-build/issues/81](https://github.com/texjporg/tex-jp-build/issues/81).
- [6] H. Kitagawa. Distinction of Latin characters and Japanese characters in stringization of pTeX family (in Japanese). [osdn.net/projects/eptex/docs/tc19ptex/ja/1/tc19ptex.pdf](https://osdn.net/projects/eptex/docs/tc19ptex/ja/1/tc19ptex.pdf).
- [7] H. Kitagawa. ε-pTeX Wiki (in Japanese). [osdn.net/projects/eptex/wiki/FrontPage](https://osdn.net/projects/eptex/wiki/FrontPage).
- [8] LATEX Project Team. LATEX news, issue 28. [www.latex-project.org/news/latex2e-news/1tnews28.pdf](https://www.latex-project.org/news/latex2e-news/1tnews28.pdf), Apr. 2018.
- [9] H. Okumura. pTeX and Japanese Typesetting. *The Asian Journal of TeX* 2(1):43–51, 2008. [ajt.ktug.org/2008/0201okumura.pdf](https://ajt.ktug.org/2008/0201okumura.pdf)
- [10] T. Tanaka. upTeX, upLaTeX — unicode version of pTeX, pLaTeX. [www.t-lab.opal.ne.jp/tex/uptex\\_en.html](https://www.t-lab.opal.ne.jp/tex/uptex_en.html).
- [11] T. Tanaka. upTeX — Unicode version of pTeX with cjk extensions. *TUGboat* 34(3):285–288, 2013. [tug.org/TUGboat/tb34-3/tb108tanaka.pdf](https://tug.org/TUGboat/tb34-3/tb108tanaka.pdf)
- [12] N. Tutimura. UTF-8 対応 (4) — ptetex Wiki (in Japanese). [tutimura.ath.cx/ptetex/?UTF-8%C2%D0%B1%FE%284%29](https://tutimura.ath.cx/ptetex/?UTF-8%C2%D0%B1%FE%284%29).

◇ Hironori Kitagawa  
Tokyo, Japan  
[h\\_kitagawa2001 \(at\) yahoo dot co dot jp](mailto:h_kitagawa2001@yahoo.co.jp)

## Keyword scanning

Hans Hagen

Some primitives in  $\text{\TeX}$  can take one or more optional keywords and/or keywords followed by one or more values. In traditional  $\text{\TeX}$  it concerns a handful of primitives, in  $\text{pdft\TeX}$  there are plenty of backend-related primitives,  $\text{Lua\TeX}$  introduced optional keywords to some math constructs and attributes to boxes, and  $\text{LuaMeta\TeX}$  adds some more too. The keyword scanner in  $\text{\TeX}$  is rather special. Keywords are used in cases like:

```
\hbox spread 10cm {...}
\advance\scratchcounter by 10
\vrule width 3cm height 1ex
```

Sometimes there are multiple keywords, as with rules, in which case you can imagine a case like:

```
\vrule width 3cm depth 1ex width 10cm depth 0ex
height 1ex\relax
```

Here we add a `\relax` to end the scanning. If we don't do that and the rule specification is followed by arbitrary (read: unpredictable) text, the next word might be a valid keyword and when followed by a dimension (unlikely) it will happily be read as a directive, or when not followed by a dimension an error message will show up. Sometimes the scanning is more restricted, as with glue where the optional `plus` and `minus` are to come in that order, but when missing, again a word from the text can be picked up if one doesn't explicitly end with a `\relax` or some other token.

```
\scratchskip = 10pt plus 10pt minus 10pt % okay
\scratchskip = 10pt plus 10pt % okay
\scratchskip = 10pt minus 10pt % okay
\scratchskip = 10pt plus whatever % error
% typesets "plus 10pt":
\scratchskip = 10pt minus 10pt plus 10pt
```

The scanner is case insensitive, so the following specifications are all valid:

```
\hbox To 10cm {To}
\hbox T0 10cm {T0}
\hbox t0 10cm {t0}
\hbox to 10cm {to}
```

It happens that keywords are always simple English words so the engine uses a cheap check deep down, just offsetting to uppercase, but of course that will not work for arbitrary UTF-8 (as used in  $\text{Lua\TeX}$ ) and it's also unrelated to the upper- and lowercase codes as  $\text{\TeX}$  knows them.

The above lines scan for the keyword `to` and after that for a dimension. While keyword scanning is case tolerant, dimension scanning is period tolerant:

```
\hbox to 10cm {10cm}
```

```
\hbox to 10.0cm {10.0cm}
\hbox to .0cm { .0cm}
\hbox to .cm { .cm}
\hbox to 10.cm {10.cm}
```

These are all valid and according to the specification; even the single period is okay, although it looks funny. It would not be hard to intercept that but I guess that when  $\text{\TeX}$  was written anything that could harm performance was taken into account. One can even argue for cases like:

```
\hbox to \first.\second cm { .cm}
```

Here `\first` and/or `\second` can be empty. Most users won't notice these side effects of scanning numbers anyway.

## Pushing back tokens

The reason for writing up any discussion of keywords is the following. Optional keyword scanning is kind of costly, not so much now, but more so decades ago (which led to some interesting optimizations, as we'll see). For instance, in the first line below, there is no keyword. The scanner sees a 1 and it not being a keyword, pushes that character back in the input.

```
\advance\scratchcounter 10
\advance\scratchcounter by 10
```

In the case of:

```
\scratchskip 10pt plux
```

it has to push back the four scanned tokens `plux`. Now, in the engine there are lots of cases where lookahead happens and when a condition is not satisfied, the just-read token is pushed back. Incidentally, when picking up the next token triggered some expansion, it's not the original next token that gets pushed back, but the first token seen after the expansion. Pushing back tokens is not that inefficient, although it involves allocating a token and pushing and popping input stacks (we're talking of a mix of reading from file, token memory,  $\text{Lua}$  prints, etc.) but it always takes a little time and memory. In  $\text{Lua\TeX}$  there are more keywords for boxes, and there we have loops too: in a box specification one or more optional attributes are scanned before the optional `to` or `spread`, so again there can be push back when no more `attr` are seen.

```
\hbox attr 1 98 attr 2 99 to 1cm{...}
```

In  $\text{LuaMeta\TeX}$  there is even more optional keyword scanning, but we leave that for now and just show one example:

```
\hbox spread 10em {\hss
\hbox orientation 0 yoffset 1mm to 2em {up}\hss
\hbox to 2em{here}\hss
\hbox orientation 0 xoffset-1mm to 2em{down}\hss
}
```

Although one cannot mess too much with these low-level scanners there was room for some optimization, so the penalty we pay for more keyword scanning in LuaMetaTeX is not that high. (I try to compensate when adding features that have a possible performance hit with some gain elsewhere.)

It will be no surprise that there can be interesting side effects to keyword scanning. For instance, using the two character keyword `by` in an `\advance` can be more efficient because nothing needs to be pushed back. The same is true for the sometimes optional equal:

```
\scratchskip = 10pt
```

Similar impacts on efficiency can be found in the way the end of a number is seen, basically anything not resolving to a number (or digit). (For these, assume a following token will terminate the number if needed; we're focusing on the spaces here.)

```
\scratchcounter 10% space not seen, ends \cs
\scratchcounter =10% no push back of optional =
\scratchcounter = 10% extra optional space gobble
\scratchcounter = 10 % efficient end of scanning
\scratchcounter = 10\relax % maybe less efficient
```

In the above examples scanning the number involves: skipping over spaces, checking for an optional equal, skipping over spaces, scanning for a sign, checking for an optional octal or hexadecimal trigger (single or double quote character), scanning the number till a non-digit is seen. In the case of dimensions there is fraction scanning as well as unit scanning too.

In any case, the equal is optional and kind of a keyword. Having an equal can be more efficient than not having one, again due to push back in case of no equal being seen. In the process spaces have been skipped, so add to the overhead the scanning for optional spaces. In LuaMetaTeX all that has been optimized a bit. By the way, in dimension scanning `pt` is actually a keyword and as there are several dimensions possible quite some push back can happen there, but we scan for the most likely candidates first.

### Catcode surprises

All that said, we're now ready for a surprise. The keyword scanner gets a string that it will test for, say, `to` in case of a box specification. It then will fetch tokens from whatever provides the input. A token encodes a so-called command and a character and can be related to a control sequence. For instance, the character `t` becomes a letter command with related value 116. So, we have three properties: the command code, the character code and the control sequence code. Now, instead of checking if the

command code is a letter or other character (two checks) a fast check happens for the control sequence code being zero. If that is the case, the character code is compared. In practice that works out well because the characters that make up a keyword are in the range 65–90 and 97–122, and all other character codes are either below that (the ones that relate to primitives where the character code is actually a sub-command of a limited range) or much larger numbers that, for instance, indicate an entry in some array, where the first useful index is above the mentioned ranges.

The surprise is in the fact that there is no checking for letters or other characters, so this is why the following code will work too:<sup>1</sup>

```
\catcode'0= 1 \hbox t0 10cm {...}% { begingroup
\catcode'0= 2 \hbox t0 10cm {...}% } endgroup
\catcode'0= 3 \hbox t0 10cm {...}% $ mathshift
\catcode'0= 4 \hbox t0 10cm {...}% & alignment
\catcode'0= 6 \hbox t0 10cm {...}% # parameter
\catcode'0= 7 \hbox t0 10cm {...}% ^ superscript
\catcode'0= 8 \hbox t0 10cm {...}% _ subscript
\catcode'0=11 \hbox t0 10cm {...}% letter
\catcode'0=12 \hbox t0 10cm {...}% other
```

In the first line, if we changed the catcode of `T` (instead of 0), it gives an error because TeX sees a begin group character (category code 1) and starts the group, but as a second character in a keyword (0) it's okay because TeX will not look at the category code.

Of course only the cases 11 and 12 make sense in practice. Messing with the category codes of regular letters this way will definitely give problems with processing normal text. In a case like:

```
{\catcode 'o=3 \hbox to 10cm {oops}} % \hb
{\catcode '0=3 \hbox to 10cm {Oeps}} % {$eps}
```

we have several issues: the primitive control sequence `\hbox` has an `o` so TeX will stop after `\hb` which can be undefined or a valid macro and what happens next is hard to predict. Using uppercase will work but then the content of the box is bad because there the `O` enters math. Now consider:

```
{\catcode '0=3 \hbox to 10cm {Oeps Oeps}}
% {$eps $eps}
```

This will work because there are now two 0's in the box, so we have balanced inline math triggers. But how does one explain that to a user? (Who probably doesn't understand where an error message comes from in the first place.) Anyway, this kind of tolerance is still not pretty, so in LuaMetaTeX we now check for the command code and stick to letters

---

<sup>1</sup> No longer in LuaMetaTeX where we do a bit more robust check.

and other characters. On today's machines (and even on my by now ancient workhorse) the performance hit can be neglected.

In fact, by intercepting the weird cases we also avoid an unnecessary case check when we fall through the zero control sequence test. Of course that also means that the above mentioned category code trickery doesn't work any more: only letters and other characters are now valid in keyword scanning. Now, it can be that some macro programmer actually used those side effects but apart from some macro hacker being hurt because no longer mastering those details can be showed off, it is users that we care more for, don't we?

### Current performance

To be sure, the abovementioned performance of keyword and equal scanning is not that relevant in practice. But for the record, here are some timings on a laptop with a i7-3849QM processor using MinGW binaries on a 64-bit Windows 10 system. The times are the averages of five times a million such assignments and advancements.

one million times	terminal	LMTX	LuaTeX
\advance\scratchctr 1	space	0.068	0.085
\advance\scratchctr 1	\relax	0.135	0.149
\advance\scratchctr by 1	space	0.087	0.099
\advance\scratchctr by 1	\relax	0.155	0.161
\scratchctr 1	space	0.057	0.096
\scratchctr 1	\relax	0.125	0.151
\scratchctr=1	space	0.063	0.080
\scratchctr=1	\relax	0.131	0.138

We differentiate here between using a space as terminal or a \relax. The latter is a bit less efficient because more code is involved in resolving the meaning of the control sequence (which eventually boils down to nothing) but nevertheless, these are not timings that one can lose sleep over, especially when the rest of a decent TeX run is taken into account. And yes, LuaMetaTeX (LMTX) is a bit faster here than LuaTeX, but I would be disappointed if that weren't the case.

◊ Hans Hagen  
<http://pragma-ade.com>

---

### Representation of macro parameters

Hans Hagen

When TeX reads input it either does something directly, like setting a register, loading a font, turning a character into a glyph node, packaging a box, or it sort of collects tokens and stores them somehow, in a macro (definition), in a token register, or someplace temporary to inject them into the input later. Here we'll be discussing macros, which have a special token list containing the preamble defining the arguments and a body doing the real work. For instance when you say:

```
\def\foo#1#2{#1 + #2 + #1 + #2}
```

the macro \foo is stored in such a way that it knows how to pick up the two arguments and when expanding the body, it will inject the collected arguments each time a reference like #1 or #2 is seen. In fact, quite often, TeX pushes a list of tokens (like an argument) in the input stream and then detours in taking tokens from that list. Because TeX does all its memory management itself the price of all that copying is not that high, although during a long and more complex run the individual tokens that make the forward linked list of tokens get scattered in token memory and memory access is still the bottleneck in processing.

A somewhat simplified view of how a macro like this gets stored is the following:

```
hash entry "foo" with property "macro call" =>
```

```
match (# property stored)
match (# property stored)
end of match

match reference 1
other character +
match reference 2
other character +
match reference 1
other character +
match reference 2
```

When a macro gets expanded, the scanner first collects all the passed arguments and then pushes those (in this case two) token lists on the parameter stack. Keep in mind that due to nesting many kinds of stacks play a role. When the body gets expanded and a reference is seen, the argument that it refers to gets injected into the input, so imagine that we have this definition:

```
\foo#1#2{\ifdim\dimen0=0pt #1\else #2\fi}
```

and we say:

```
\foo{yes}{no}
```

then it's as if we had typed:

```
\ifdim\dimen0=0pt yes\else no\fi
```

So, you'd better not have something in the arguments that messes up the condition parser! From the perspective of an expansion machine it all makes sense. But it also means that when arguments are not used, they still get parsed and stored. Imagine using this one:

```
\def\foo#1{\iffalse#1\oof#1\oof#1\oof#1\oof#1\fi}
```

When TeX sees that the condition is false it will enter a fast scanning mode where it only looks at condition related tokens, so even if \oof is not defined this will work ok:

```
\foo{!}
```

But when we say this:

```
\foo{\else}
```

it will bark! This is because each #1 reference will be resolved, so we effectively have (line breaks in the following are editorial)

```
\def\foo#1{\iffalse\else\oof\else\oof\else\oof
 \else\oof\else\fi}
```

which is not good. On the other hand, since no expansion takes place in quick parsing mode, this will work:

```
\def\oof{\else}
\foo\oof
```

which to TeX looks like:

```
\def\foo#1{\iffalse\oof\oof\oof\oof\oof\oof
 \oof\oof\fi}
```

So, a reference to an argument effectively is just a replacement. As long as you keep that in mind, and realize that while TeX is skipping ‘if’ branches nothing gets expanded, you’re okay.

Most users will associate the # character with macro arguments or preambles in low level alignments, but since most macro packages provide a higher level set of table macros the latter is less well known. But, as often with characters in TeX, you can do magic things:

```
\catcode'?=\\catcode'#
\def\foo #1#2?3{?1?2?3}
 \meaning\foo\space=>\foo{1}{2}{3}\par
\def\foo ?1#2?3{?1?2?3}
 \meaning\foo\space=>\foo{1}{2}{3}\par
\def\foo ?1?2#3{?1?2?3}
 \meaning\foo\space=>\foo{1}{2}{3}\par
```

Here the question mark also indicates a macro argument. However, when expanded we see this as result:

```
macro:#1#2?3->?1?2?3 =>123
macro:?1#2?3->?1?2?3 =>123
macro:?1?2#3->#1#2#3 =>123
```

The last used argument signal character (officially called a match character, here we have two that fit that category, # and ?) is used in the serialization! Now, there is an interesting aspect here. When TeX stores the preamble, as in our first example:

```
match (# property stored)
match (# property stored)
end of match
```

the property is stored, so in the later example we get:

```
match (# property stored)
match (# property stored)
match (? property stored)
end of match
```

But in the macro body the number is stored instead, because we need it as reference to the parameter, so when that bit gets serialized TeX (or more accurately: LuaTeX, which is what we’re using here) doesn’t know what specific signal was used. When the preamble is serialized it does keep track of the last so-called match character. This is why we see this inconsistency in rendering.

A simple solution would be to store the used signal for the match argument, which probably only takes a few lines of extra code (using a nine integer array instead of a single integer), and use that instead. I’m willing to see that as a bug in LuaTeX but when I ran into it I was playing with something else: adding the ability to prevent storing unused arguments. But the resulting confusion can make one wonder why we do not always serialize the match character as #.

It was then that I noticed that the preamble stored the match tokens and not the number and that TeX in fact assumes that no mixture is used. And, after prototyping that in itself trivial change I decided that in order to properly serialize this new feature it also made sense to always serialize the match token as #. I simply prefer consistency over confusion and so I caught two flies in one stroke. The new feature is indicated with a \#0 parameter:

```
\bgroup
\catcode'?=\\catcode'#
\def\foo ?1#0?3{?1?2?3}
 \meaning\foo\space=>\foo{1}{2}{3}\crlf
\def\foo ?1#0?3{?1?2?3}
 \meaning\foo\space=>\foo{1}{2}{3}\crlf
\def\foo #1#2?3{?1?2?3}
 \meaning\foo\space=>\foo{1}{2}{3}\crlf
\def\foo ?1#2?3{?1?2?3}
 \meaning\foo\space=>\foo{1}{2}{3}\crlf
\def\foo ?1?2#3{?1?2?3}
 \meaning\foo\space=>\foo{1}{2}{3}\crlf
\def\foo ?1?2#3{?1?2?3}
 \meaning\foo\space=>\foo{1}{2}{3}\crlf
\egroup
```

shows us:

```
macro:#1#0#3->#1#2#3 =>13
macro:#1#0#3->#1#2#3 =>13
macro:#1#2#3->#1#2#3 =>123
macro:#1#2#3->#1#2#3 =>123
macro:#1#2#3->#1#2#3 =>123
```

So, what is the rationale behind this new #0 variant? Quite often you don't want to do something with an argument at all. This happens when a macro acts upon for instance a first argument and then expands another macro that follows up but only deals with one of many arguments and discards the rest. Then it makes no sense to store unused arguments. Keep in mind that in order to use it more than once an argument does need to be stored, because the parser only looks forward. In principle there could be some optimization in case the tokens come from macros but we leave that for now. So, when we don't need an argument, we can avoid storing it and just skip over it. Consider the following:

```
\def\foo#1{\ifnum#1=1 \expandafter\foooone
 \else\expandafter\footwo\fi}
\def\foooone#1#0{#1}
\def\footwo#0#2{#2}
\foo{1}{yes}{no}
\foo{0}{yes}{no}
```

We get:

yes no

Just for the record, tracing of a macro shows that indeed there is no argument stored:

```
\def\foo#1#0#3{....}
\foo{11}{22}{33}
\foo #1#0#3->....
#1<-11
#2<-
#3<-33
```

Now, you can argue, what is the benefit of not storing tokens? As mentioned above, the  $\text{\TeX}$  engines do their own memory management.<sup>1</sup> This has large benefits in performance especially when one keeps in mind that tokens get allocated and are recycled constantly (take only lookahead and push back).

However, even if this means that storing a couple of unused arguments doesn't put much of a dent in performance, it does mean that a token sits somewhere in memory and that this bit of memory needs to get accessed. Again, this is no big deal on a computer where a  $\text{\TeX}$  job can take one core and basically is the only process fighting for CPU cache usage. But less memory access might be more relevant in a scenario of multiple virtual machines running on the same hardware or multiple  $\text{\TeX}$  processes on one

---

<sup>1</sup> An added benefit is that dumping and undumping is relatively efficient too.

machine. I didn't carefully measure that so I might be wrong here. Anyway, it's always good to avoid moving around data when there is no need for it.

Just to temper expectations with respect to performance, here are some examples:

```
\catcode`!=9 % ignore this character
\firsof twoarguments
{!!!!!!!!!!!!!!}{!!!!!!!!!!!!!!}
\secondoftwoarguments
{!!!!!!!!!!!!!!}{!!!!!!!!!!!!!!}
\secondoffourarguments
{!!!!!!!!!!!!!!}{!!!!!!!!!!!!!!}
{!!!!!!!!!!!!!!}{!!!!!!!!!!!!!!}
```

In Con $\text{\TeX}$  we define these macros as follows:

```
\def\firsof twoarguments #1#2{#1}
\def\secondoftwoarguments #1#2{#2}
\def\secondoffourarguments#1#2#3#4{#2}
```

The performance of 2 million expansions is the following (probably half or less on a more modern machine):

macro	total	step
\firsof twoarguments	0.245	0.000000123
\secondoftwoarguments	0.251	0.000000126
\secondoffourarguments	0.390	0.000000195

But we could use this instead:

```
\def\firsof twoarguments #1#0{#1}
\def\secondoftwoarguments #0#2{#2}
\def\secondoffourarguments#0#2#0#0{#2}
```

which gives:

macro	total	step
\firsof twoarguments	0.229	0.000000115
\secondoftwoarguments	0.236	0.000000118
\secondoffourarguments	0.323	0.000000162

So, no impressive differences, especially when one considers that when that many expansions happen in a run, getting the document itself rendered plus expanding real arguments (not something defined to be ignored) will take way more time compared to this. I always test an extension like this on the test suite<sup>2</sup> as well as the LuaMeta $\text{\TeX}$  manual (which takes about 11 seconds) and although one can notice a little gain, it makes more sense not to play music on the same machine as we run the  $\text{\TeX}$  job, if gaining milliseconds is that important. But, as said, it's more about unnecessary memory access than about CPU cycles.

This extension is downward compatible and its overhead can be neglected. Okay, the serialization

---

<sup>2</sup> Currently some 1600 files that take 24 minutes plus or minus 30 seconds to process on a high end 2013 laptop. The 260 page manual with lots of tables, verbatim and MetaPost images takes around 11 seconds. A few milliseconds more or less don't really show here. I only time these runs because I want to make sure that there are no dramatic consequences.

now always uses # but it was inconsistent before, so I'm willing to sacrifice that (and I'm pretty sure no ConTEXt user cares or will even notice). Also, it's only in LuaMetaTEX (for now) so that other macro packages don't suffer from this patch. The few cases where ConTEXt can benefit from it are easy to isolate for MkIV and LMTX so we can support LuaTEX and LuaMetaTEX.

I mentioned LuaTEX and how it serializes, but for the record, let's see how pdfTEX, which is very close to original TEX in terms of source code, does it. If we have this input:

```
\catcode`D=\catcode`#
\catcode`0=\catcode`#
\catcode`N=\catcode`#
\catcode`-=\catcode`#
\catcode`K=\catcode`#
\catcode`N=\catcode`#
\catcode`U=\catcode`#
\catcode`T=\catcode`#
\catcode`H=\catcode`#
\def\dek D102N3-4K5N6U7T8H9{#1#2#3 #4#5#6#7#8#9}
{\meaning\dek \tracingall \dek don{}knuth}
```

The meaning gets typeset as (again, line break is editorial):

```
macro:D102N3-4K5N6U7T8H9->H1H2H3 H4H5H6H7
H8H9don knuth
```

while the tracing reports:

```
\dek D102N3-4K5N6U7T8H9->H1H2H3 H4H5H6H7H8H9
D1<-d
O2<-o
N3<-n
-4<-
K5<-k
N6<-n
U7<-u
T8<-t
H9<-h
```

The reason for the difference, as mentioned, is that the tracing uses the template and therefore uses the stored match token, while the meaning uses the reference match tokens that carry the number and at that time has no access to the original match token. Keeping track of that for the sake of tracing would not make sense anyway. So, traditional TEX, which is what pdfTEX is very close to, uses the last used match token, the H. Maybe this example can convince you that dropping that bit of log related compatibility is not that much of a problem. I just tell myself that I turned an unwanted side effect into a new feature.

## A few side notes

The fact that characters can be given a special meaning is one of the charming properties of TEX. Take these two cases:

```
\bgroup\catcode`\&=5 &\egroup
\bgroup\catcode`\!=5 !\egroup
```

In both lines there is now an alignment character used outside an alignment. And, in both cases the error message is similar:

```
! Misplaced alignment tab character &
! Misplaced alignment tab character !
```

So, indeed the right character is shown in the message. But, as soon as you ask for help, there is a difference: in the first case the help is specific for a tab character, but in the second case a more generic explanation is given. Just try it.

The reason is an explicit check for the ampersand being used as tab character. Such is the charm of TEX. I'll probably opt for a trivial change to be consistent here, although in ConTEXt the ampersand is just an ampersand so no user will notice.

There are a few more places where, although in principle any character can serve any purpose, there are hard coded assumptions, like \$ being used for math, so a missing dollar is reported, even if math started with another character being used to enter math mode. This makes sense because there is no urgent need to keep track of what specific character was used for entering math mode. An even stronger argument could be that TEXies expect dollars to be used for that purpose. Of course this works fine:

```
\catcode`\$=\catcode`\$
\$ \sqrt{x^3} \$
```

But when we forget an \$ we get messages like:

```
! Missing $ inserted
```

or more generic:

```
! Extra }, or forgotten $
```

which is definitely a confirmation of “America first”. Of course we can compromise in display math because this is quite okay:

```
\catcode`\$=\catcode`\$
\$ \sqrt{x^3} \$
```

unless of course we forget the last dollar in which case we are told that

```
! Display math should end with $$
```

so no matter what, the dollar wins. Given how ugly the Euro sign looks I can live with this, although I always wonder what character would have been taken if TEX was developed in another country.

## TeXdoc online—a web interface for serving TeX documentation

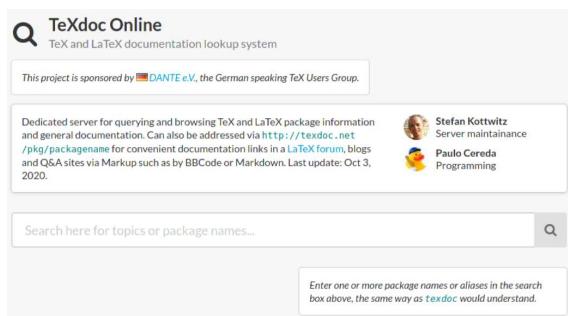
Island of TeX (developers)

### Abstract

When looking for TeX-related documentation, users have many options, including running `texdoc` on their local machine, looking up the package at CTAN, or using a service like `texdoc.net`. As the latter is known for lacking regular updates, the Island of TeX decided to take the opportunity to provide a complete rewrite of the provided service using a RESTful API and a self-updating Docker container.

### 1 Core features of `texdoc.net`

The most important feature of `texdoc.net` is the documentation search as the prominent first item on the landing page (cf. fig. 1). Searching for a package yields a table with the entries a user could retrieve locally by executing `texdoc -l <package>`. Each entry is linked to the corresponding document and the user is able to either view (if there is browser view support for the file type) or download the documentation file. This is especially useful for users without a local TeX installation like Overleaf users.



**Figure 1:** Screenshot of (top portion of) `texdoc.net`

A little bit more direct is the use of the HTTP endpoint `https://texdoc.net/pkg/<package>`; this responds with the file that `texdoc <package>` would open if executed locally. The user does not get to choose what to open but in most cases `texdoc` is good at determining the proper documentation if given the package name, so users get what they want.

The aforementioned feature of creating simple references to the documentation is what makes the service well suited for writing posts on web forums or sites like the TeX StackExchange. Linking to CTAN mirrors gives far longer urls with many unnecessary path components compared to the short syntax of `texdoc.net`.

The last noteworthy feature allows users to browse packages by topics. The list is retrieved from the (unmaintained) `texdoctk.dat` file available in the underlying TeX Live distribution. Within each topic a selection of packages is shown with the name, description and link to the main documentation for each. The topics and presented packages are not exhaustive and many packages on CTAN or even in TeX Live will not be presented to the user.

### 2 Providing a RESTful API for `texdoc`

A RESTful API is a stateless interface to web applications responding to typical HTTP requests with uniform responses. Usually, JSON is used for the response format. Following this principle, our software responds to HTTP GET requests with JSON representing documentation-related objects.

The endpoints you can access are described as follows. Keep in mind that these requests will return either HTTP status code 200 (OK) or, in the case of any error, HTTP status code 422 (Unprocessable Entity). The only endpoint that is guaranteed not to fail is located at `/version`.

`/version` This endpoint returns the versions of the API and data format (`api`), the installed version of `texdoc` (`texdoc`) and the date of the last TeX Live update as an ISO date string (`tlpdb`). Make sure your client software always expects the correct API version to avoid problems. Our API versions follow semantic versioning with all its consequences.

`/texdoc/<name>` On this path, the list of entries is returned that a local call to `texdoc -l` would produce. For each entry, there are two fields: `path`, containing the path to the documentation file relative to the `doc` subfolder of the `TEXMFDIST` directory; and `description` containing the file's description if there is one (empty otherwise). The application will always return a JSON array of such entries.

`/serve/<name>/<index>` This call results in the documentation file at index `<index>` of the result of `/texdoc/<name>` being returned to the client.

`/pkg/<name>` This endpoint is a shortcut for the `/serve/<name>/0` endpoint, defined to preserve compatibility with the API of `texdoc.net`.

`/topics/list` This endpoint returns the list of topics known to the application specified by their key and a caption called `details`. This is a direct interface to CTAN's API for topics. Network access for the server software is required.

`/topic/<name>` This endpoint returns details for a topic by returning the key (what is passed as

`(name)`), a string with a short description called `details` and a list of package names (strings) called `packages`. This is a direct interface to CTAN’s API for topics. Network access for the server software is required.

### 3 The new front end

The front end of `TeXdoc` online is structured in a similar way to `texdoc.net`. The main feature is still searching for packages. This is based on the `/texdoc/<package>` endpoint presented in the previous section. The results will be the same as on `texdoc.net`.

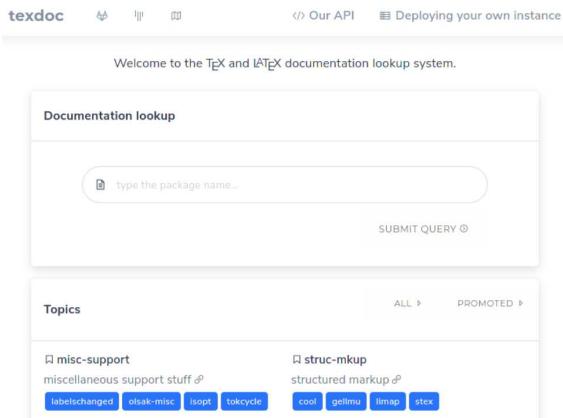


Figure 2: Screenshot of (top portion of) `texdoc-online`

Topics are handled differently, though. We use CTAN’s JSON API to fetch their topics and packages belonging to these topics. Any user visiting the landing page will be shown six random categories with a few packages each. If a category holds several packages, four of them are selected at random. Users have the option to show all topics and list the packages for any topic they are interested in. Also, each package entry can be automatically queried for documentation. This covers many more packages than the old `texdoctk.dat` file, though it does have the disadvantage that some of CTAN’s topics are sparsely populated.

The front page also offers a “jump to documentation” section introducing the API and options to host the software yourself. These are covered in the next section.

### 4 Deploying the software

The source code for `TeXdoc` online is hosted at GitLab and may be found at <https://gitlab.com/islandoftex/images/texdoc-online>. There are also download options to get an executable JAR file with all dependencies included that you can simply

run using your local Java installation. Running the JAR bundle will open a local web server that can be accessed using the url `http://127.0.0.1:8080` from your browser. It will not work without Internet access (because of how it fetches topics) or a local `TeX` installation (for `TeXdoc`) though.

As an alternative to running the JAR file we also provide a dockerized image that users can pull using (line break is editorial; this is a one-line string)  
`registry.gitlab.com/islandoftex/images/texdoc-online:latest`

This image is based on our `TeX` Live images, which makes it quite large in terms of file size but also eliminates the need for a local `TeX` installation. Using Docker is the preferred solution for hosting your own instance of `TeXdoc` online.

The Docker image also provides daily updates. The container will update daily at midnight (container time) and thus stays up to date as long as the current version of `TeX` Live is supported. Our continuous integration ensures that you can always pull the `latest` tag and receive the latest and greatest `TeX` documentation, which when pulled and run will update itself.

To ease deployment we provide a ready-made `docker-compose.yml` configuration for Docker Compose (<https://docs.docker.com/compose/>), an orchestration tool for Docker containers. It uses the Caddy web server to provide automatic HTTPS with local certificates or, on public domains, Let’s Encrypt certificates. Alter the `localhost` line within that file to suit your needs and `docker-compose up` will start a production-ready instance of `TeXdoc` online.

### 5 Final remarks

In cooperation with Stefan Kottwitz we are working on providing a hosted solution as a future replacement of `texdoc.net`. Stay tuned for seeing this transition happen within the next months.

`TeXdoc` online is still a work in progress and there is room for improvement. We are working on new features as well as considering ways to extend the current front-end for additional, hosting-based content. The RESTful API, through the endpoints presented in section 2, allows external applications and services to easily query `TeX` package documentation from an updated lookup system.

Feedback can be provided through the project repository in GitLab. We look forward to hearing from you!

◊ Island of `TeX` (developers)  
<https://gitlab.com/islandoftex>

---

## MMTeX: Creating a minimal and modern TeX distribution for GNU/Linux

Michal Vlasák

MMTeX stands for “minimal modern TeX” and is a simple, small and legacy-free distribution of TeX for GNU/Linux. It has the form of a installable package, offers full functionality of OpTeX and plain LuaTeX formats, allows use of system fonts and resources in external TEXMF trees.

This article explains my motivation for creating it, describes some aspects of a distribution in general and how they are handled in MMTeX. My goal is to show that things can be done simply, and that TeX can integrate better into a Unix system and not be the odd one out.

### Motivation

I find TeX Live huge and complicated. Its “full scheme” installation, which is the default, takes up about 7 GiB. Though with “minimal” (plain TeX only) scheme and no documentation you can get to about 50 MiB. I also think that it doesn’t fit very neatly into a Unix system. It isolates itself more or less in a single directory tree and doesn’t follow the hier(7) standard; as two examples, it doesn’t store configuration files in /etc and doesn’t permit read-only mount of /usr.

My end goal was to create something that:

- is a single package that can be installed using an operating system’s package manager,
- integrates well into a system (respecting filesystem hierarchy, using system fonts),
- includes only core functionality, but could easily be pointed to external TEXMF tree(s) with additional packages/files,
- doesn’t complicate things with legacy, dynamic regeneration of various files, ...

The intended users are those who want a small (34 MiB) but functional TeX system, one comparable to L<sup>A</sup>T<sub>E</sub>X and its many packages. Because of its small size and ability to install using a package manager it can also be useful for Docker images or CI/CD scripts that need to set up a TeX environment.

### Engine

Nowadays support for PDF output and OpenType fonts is a must. That leaves the choice of engine between LuaTeX and XeTeX. LuaTeX was chosen, because it is extensible with Lua, has better support of micro-typographical extensions, and has integrated METAPOST in the form of `mplib`.

**Compiling LuaTeX.** Originally LuaTeX started with pdftEX’s sources (mostly written in WEB), but was translated to C. LuaTeX is somewhat confusingly developed in a repository that is essentially a subset of TeX Live’s repository, but separate. TeX Live’s build infrastructure is based on GNU Auto-tools and is able to compile all of TeX Live and external libraries for *a lot* of platforms, but is very slow and does a lot of checks for things that have been standardized in C or POSIX for years.

After preparing replacements of build-time generated files it is however possible to compile LuaTeX in more or less a single call to the compiler—the only catch is `mplib`. It is written in CWEB, which itself is written in CWEB, so bootstrapping is needed.

**External libraries.** LuaTeX uses several external C libraries. The most prominent one is of course Lua, but it also uses, for example, `libpng` to handle PNG images. There are two choices for how to use a library—either compile its source into the binary (“statically link”) or on each run of the program find and load the compiled library file somewhere on the file system (“dynamically link”). The usual choice for most systems is dynamic linking—this allows reuse of a single library file for more programs (making updates easier) and saves disk space. It is a bit slower, because of the searching and loading.

The `libpng` and `zlib` libraries, for example, are often already present on systems or can easily be installed using a package manager. For these the dynamic linking approach is better. Other libraries have LuaTeX’s modifications (Lua) or are specific to TeX (Kpathsea) so sharing them would not be especially useful. These are statically linked.

### Formats

The obvious choice of format for a minimal TeX would be plain TeX. Or rather its adaptation for LuaTeX which for example outputs to PDF by default. While it can be called minimal, it isn’t “modern”. Most users today expect a format to be able to easily create documents with numbered sections, tables of contents, bibliographies, hyperlinks, source code listings with syntax highlighting, etc. A recent format, based on LuaTeX, which provides these features, but still keeps plain’s simplicity is OpTeX. In a sense it is even more powerful than L<sup>A</sup>T<sub>E</sub>X—where L<sup>A</sup>T<sub>E</sub>X needs a package or an external binary, OpTeX has it built in.

Both formats are included in the distribution, OpTeX is the primary one, while plain is for now included more or less just to allow running `luatex` without getting an error about a missing format.

## Finding files

`LuaTeX` uses the Kpathsea library for finding files. Kpathsea uses path specifications and variables similar to the Unix PATH environment variable, but differentiates between file types. For each file type it maintains one or more associated variable names, a list of possible suffixes and most importantly a calculated *search path* (directories separated by colons). For example `bib_format`'s variables are `BIBINPUTS` and `TEXBIB`, while the suffix list contains `.bib`.

In a simple case a search path is determined in one of three ways, in order of significance:

- value of associated variable set in the environment (that is, an environment variable),
- value of associated variable from a `texmf.cnf` configuration file,
- default value set at compilation time.

For finding `.cnf` files the same path searching mechanism is used; the variable is `TEXMFCONF`, but as it cannot be set from a configuration file, only the first and last way applies. All `texmf.cnf` files that are found are read. Order is important—earlier assignments in configuration files override later ones.

I set all the useful file types to have defaults that work without any configuration file, and respect standards like `hier(7)`, TDS and XDG. For example, the search path for `.cnf` files is:

- `TEXMFDOTDIR` (more on this later),
- `~/.config/mmtex` (or more precisely its XDG equivalent),
- `/etc/mmtex`,

to allow local (“project”), user and system configuration files respectively.

In Kpathsea, default (compile-time) values for *search paths* can be set, but not *variables*. For this I created a patch that “injects” default values for a few variables, as if they were read from a configuration file.

`TEXMFDOTDIR` variable was inspired by `TeX Live` and is normally the current directory (“.”), but is useful for temporary overrides on the command line, using environment variables. Every search path contains `TEXMFDOTDIR` as the first entry, even the one for `.cnf` files (allowing for project-specific settings).

`TEXMF` is the most important variable for `MMTeX`. It should contain roots of all `TEXMF` trees. It is supposed to be set by the user or system administrator at any level of configuration they need at the moment, and doesn't have a default value (preferences of users and system administrators vary widely).

## Language support

Previous `TeX` engines had the limitation of being able to load hyphenation patterns only at format creation time—when running `initTeX`. `LuaTeX` has no such limitation; by using Lua, it is possible to load hyphenation patterns at runtime.

Today virtually all hyphenation patterns and exceptions that have been used by `TeX` users are distributed in the `hyph-utf8` package. `hyph-utf8` also provides patterns and exceptions in UTF-8 encoded text files, which are preferred for `LuaTeX`.

`TeX Live`'s approach is to provide hyphenation patterns and exceptions for each language in a separate package. Each package then hooks itself using the `TeX Live` `execute AddHyphen` directive. An example for French:

```
execute AddHyphen \
 name=french synonyms=patois,francais \
 lefthyphenmin=2 righthyphenmin=2 \
 file=loadhyph-fr.tex \
 file_patterns=hyph-fr.pat.txt \
 file_exceptions=
```

This information is also written to files used by `ε-TeX`'s language mechanism, which is used by plain `LuaTeX`. This gets added to `language.def`:

```
\addlanguage{french}{loadhyph-fr.tex}{}{2}{2}
```

and this is written to `language.dat.lua`:

```
[‘french’] = {
 loader = ‘loadhyph-fr.tex’,
 lefthyphenmin = 2,
 righthyphenmin = 2,
 synonyms = { ‘patois’, ‘francais’ },
 patterns = ‘hyph-fr.pat.txt’,
 hyphenation = ‘’, },
```

`etex.src` reads `language.def` at format creation time. Listed languages are registered and their hyphenation patterns loaded into the format. This enables their use later with `\uselanguage`.

In `LuaTeX`, it is discouraged to load patterns into the format, so the mechanism is changed by `hyph-utf8`'s own `etex.src`. Instead of loading each pattern or exception file on `\addlanguage`, the language is only registered and the files are loaded at the first `\uselanguage`. Both commands use Lua code in `luatex-hyphen.lua`, which uses information in `language.dat.lua` for handling synonyms and finding the names of pattern files.

In `OpTeX` the situation is simpler. It doesn't read `language.def` because it already has that information, but it still uses `luatex-hyphen.lua`.

To support all languages in `hyph-utf8`, `MMTeX` generates the files `language.def` and `language.dat.lua` from the `hyph-utf8` sources.

## Fonts

To fully use the potential of LuaTeX, OpenType fonts should be used. These are the same fonts that are used by other programs, and as such some of them are already preinstalled on operating systems. And probably many more are additionally installed by users or administrators. To also not duplicate any effort with packaging of fonts, the distribution doesn't provide any OpenType fonts. The idea is to let users use the fonts they already have or can get on their system, as well as the fonts they have in their TEXMF tree(s). For example Latin Modern, the GUST e-foundry adaptation of Computer Modern which includes OpenType, is available as `fonts-lmodern` on Debian-based systems and `otf-latin-modern` on Arch Linux.

**8-bit fonts.** Only 8-bit fonts can be preloaded into a TeX format. Both OpTeX and plain LuaTeX do this. To support this, MMTeX includes a minimal set of Type 1 fonts and their respective metric and encoding files. A `pdftex.map` file is needed, as it is used to map names of TFM metric files to font names and font files, with optional reencodings. This file contains lines like:

```
cmr5 CMR5 <cmr5.pfb
ec-lmr5 LMRoman5-Regular <lm-ec.enc <lmr5.pfb
```

The first line connects the `cmr5.tfm` font metric file, the `cmr5.pfb` Type 1 font and the CMR5 font name inside the `.pfb`. (CMR5 stands for Computer Modern Roman in 5 point optical size). The second line is similar, but additionally refers to a so-called encoding vector stored in file `lm-ec.enc`. This is necessary because `lmr5.pfb` contains many glyphs, while TeX can use only 256 of them and expects the order to correspond with `ec-lmr5.tfm`, which contains metric information for those selected 256 glyphs. In this particular case the Cork ("EC") encoding (set of glyphs) is used.

Engines only read one `pdftex.map` file, but each font package usually provides one or more `.map` files. This is why an aggregate `pdftex.map` is usually generated (in TeX Live using the `updmap` script). As MMTeX supports only a limited number of Type 1 fonts, a minimal `pdftex.map` was created by hand.

**OpenType fonts.** In order to handle OpenType fonts Lua code is needed. `luactfload` is included for this purpose as it is already used internally by OpTeX. It can also be used from plain LuaTeX, with `\input luactfload.sty`.

`luactfload` is able to find all system fonts, because it reads `fontconfig`'s configuration. Therefore, there is no need to set the `OSFONTPDIR` variable.

The standard TDS directories for font files also work:  
`$TEXMF/fonts/{opentype,truetype}`.

## MetaPost

METAPOST is integrated into LuaTeX as `mplib` and available via a Lua interface. `luamplib` adapts the code from ConTeXt for plain (and LATEX), making it possible to use METAPOST in a `.tex` file. To use it, `\input luamplib.sty`.

`mplib` proved to be useful even as a METAFONT replacement. "Ralph Smith's Formal Script" font (required by OpTeX) doesn't have prebuilt TFM files on CTAN. Normally one would use METAFONT to generate the metrics, but with a few lines of Lua `mplib` can, just like METAPOST, use the `mfplain.mp` format to function as METAFONT and do the job.

## Implementation of MMTeX

MMTeX itself is a few supporting files and a script called `build` which contains instructions for building MMTeX to a given directory. The script in this form allows a wrapper that packs together the directory and some metadata to create an installable package. The included `package-builder` script demonstrates this, and as of now can create packages in Debian's `.deb`, Arch Linux's `pkg` and classic tarball (`.tar.gz`) formats.

The sources (which are merely the build logic), documentation, and prebuilt packages are available at <https://github.com/vlasakm/mmtex>.

## The result

The resulting package, installed, takes up 34 MiB (around 15 MiB compressed). Most of this is Type 1 fonts (11 MiB), the `luatex` binary (6.5 MiB), data related to Unicode codepoints (about 5 MiB) and hyphenation patterns/exceptions (3.2 MiB).

Not included are macro source files (`.dtx`) and package documentation, both of which are of course available on CTAN. Documentation is also easily accessible on <https://texdoc.net>.

At present there is no support for getting or managing packages from CTAN—MMTeX expects to be pointed to already-prepared TEXMF trees. OpTeX hopefully provides enough functionality to not require a large number of other macros.

◇ Michal Vlasák  
 Probošťov, Czech Republic  
`lahcim8 (at) gmail dot com`

## UTF-8 installations of TeX

Igor Liferenko

### Abstract

In its design TeX has the concepts of “internal encoding” and “external encoding”. This fact allows TeX to work with any encoding.

We use Unicode as TeX’s external encoding. Then we change the necessary parts of TeX to use UTF-8 as the input/output encoding.

The resulting implementation passes the `trip` test.

### 1. Initialization

Note: we use the `web2w` [1] implementation of TeX, but the ideas described here can be applied to any implementation.

First, we change the data type of characters in text files to `wchar_t` to accommodate Unicode values.

Background: this predefined C type allocates four bytes per character (on most systems). Character constants of this type are written as `L'...'` and string constants as `L"..."`.

(For brevity, in the diffs following, the original code from `web2w`’s `ctex.w` source is preceded with `<` characters, and the new code with `>`. Both are sometimes reformatted for presentation in this article, and for readability we sometimes leave a blank line between the pieces. The actual implementation uses the file `utex.patch` [2].)

```
< @d text_char unsigned char
> @d text_char wchar_t
```

Use values from the basic multilingual plane (BMP) of Unicode.

```
< @d last_text_char 255
> @d last_text_char 65535
```

Then we change the size of the `xord` array [3] to  $2^{16}$  bytes.

```
< ASCII_code xord[256];
> ASCII_code xord[65536];
```

Elements in the `xchr` array [3] are overridden using the file `mapping.w`.

```
@i mapping.w
```

This file specifies the character(s) required for a particular installation of TeX, for example:

```
xchr[0xf1] = L'ë';
```

A complete example of `mapping.w` is here:

<https://github.com/igor-liferenko/cweb>

`TEX_format_default` is in TeX’s external encoding.

```
< ASCII_code TEX_format_default
< [1+format_default_length+1]
< =" TeXformats/plain.fmt";
>
> wchar_t TEX_format_default
> [1+format_default_length+1]
> =L" TeXformats/plain.fmt";
```

It remains to set the `LC_CTYPE` locale category, on which depends the behavior of the C library functions used below

```
setlocale(LC_CTYPE, "C.UTF-8");
```

and to add the necessary headers.

```
#include <wchar.h>
#include <locale.h>
```

### 2. Input

For automatic conversion from UTF-8 to Unicode, text files (including the terminal) must be read with the C library function `fgetwc` [4].

In `ctex.w` the macro `get` is used for reading text files, as well as font metric and format files.

Text files are read in the functions `a_open_in` and `input_ln`. In `a_open_in` we replace the macro `reset` with its expansion and then in both functions we change `get(*f)` to `(*f).d=fgetwc((*f).f)`

### 3. Output

For automatic conversion from Unicode to UTF-8, text files (including the terminal) must be written with the C library function `fwprintf` [4].

In `ctex.w` the macro `write` is used for writing text files in all cases but one. So, we change `fprintf` to `fwprintf` in the definition of `write`. The one case where `write` is used is for writing DVI files—there we just use its old expansion.

In addition to redefining the macro `write`, we need to add the ‘`L`’ prefix to strings which are used in the macros that call the macro `write`. These changes are trivial and there are quite a few of them so we will not list them here. Instead, we show the following cases, where the conversion specifier in the `printf`-style directives also needs to change:

```
< wterm("%c",xchr[s]);
> wterm(L"%lc",xchr[s]);
< wlog("%c",xchr[s]);
> wlog(L"%lc",xchr[s]);
```

```
< write(write_file[selector], "%c", xchr[s]);
> write(write_file[selector], L"%lc", xchr[s]);
```

#### 4. The file name buffer

The name of the file to be opened, which is stored in the *name\_of\_file* buffer, must be encoded in UTF-8. Therefore, each character passed to *append\_to\_name*, before being added to *name\_of\_file*, must be converted to UTF-8. This is done using the C library function *wctomb* [4].

In the *append\_to\_name* macro, the variable *k* is used as the index into the *name\_of\_file* buffer where the last byte was stored. Originally, *k* was always increased and provisions were made that characters will not be written beyond the end of buffer (which has the index *file\_name\_size*); *name\_length* was then set to the minimal value between *k* and *file\_name\_size*.

We cannot do the same in our implementation, because we may reach the end of the buffer in the midst of a multibyte character. Instead, if the next multibyte character does not fit into the buffer, we prevent *k* from being increased by negating its value:

```
< @d append_to_name(X) { c=X; incr(k);
< if (k <= file_name_size)
< name_of_file[k]=xchr[c]; }

> @d append_to_name(X) { c=X;
> if (k >= 0) { /* try to append? */
> char mb[MB_CUR_MAX];
> int len = wctomb(mb, xchr[c]);
> if (k+len <= file_name_size)
> for (int i = 0; i < len; i++)
> name_of_file[++k] = mb[i];
> else
> k = -k; /* freeze k */ } }
```

In *pack\_file\_name* and *pack\_buffered\_name* (the functions that call *append\_to\_name*), we have to “unfreeze” its value if it was “frozen”.

```
if (k < 0) k = -k;
```

In *make\_name\_string*, each (multibyte) character from *name\_of\_file* must be converted from UTF-8 to Unicode, before being converted to TeX’s internal encoding. This is done using the C library function *mbtowc* [4].

```
< append_char(xord[name_of_file[k]]);

> { wchar_t wc;
> k += mbtowc(&wc, name_of_file+k,
> MB_CUR_MAX) - 1;
> append_char(xord[wc]); }
```

In the code checking *format\_default\_length* for consistency, we use the C library function *wcstombs* [4] to count the number of bytes in the UTF-8 representation of *TEX\_format\_default*.

```
< if (format_default_length >
< file_name_size)

> if (wcstombs(NULL,
> TEX_format_default+1,0) >
> file_name_size)
```

In the function *pack\_buffered\_name*, the code that drops excess characters assumes that each character is one byte:

```
if (n+b-a+1+format_ext_length >
 file_name_size)
 b=a+file_name_size-n-1-format_ext_length;
```

But the number of bytes used to represent a character in UTF-8 may be more than one. Therefore, we use an equivalent method to drop excess characters, the one which will work with multibyte characters: After appending the contents of *buffer[a .. b]* to *name\_of\_file*, we roll back in it character by character until the format extension fits in it. We use the C library function *mblen* [4] to determine the start of the next (multibyte) character to be discarded.

```
while (k+wcstombs(NULL,TEX_format_default+
 format_default_length-
 format_ext_length+1,0) >
 file_name_size) {
 k--;
 while (mblen(name_of_file+k+1,MB_CUR_MAX)
 ===-1)
 k--;
}
```

#### References

- [1] Ruckert, Martin. WEB to cweb.  
[mruckert.userweb.mwn.de/hint/web2w.html](http://mruckert.userweb.mwn.de/hint/web2w.html)
- [2] Source of the present implementation.  
<https://github.com/igor-liferenko/tex>
- [3] Knuth, Donald E. *TeX: The Program*, 1986.  
ISBN 0201134373.
- [4] Single Unix Specification. Introduction to ISO C Amendment 1 (Multibyte Support Environment).  
[http://unix.org/version2/whatsnew/login\\_mse.html](http://unix.org/version2/whatsnew/login_mse.html)

◇ Igor Liferenko  
igor.liferenko (at) gmail dot com

## OpTeX — A new generation of Plain TeX

Petr Olšák

### Introduction

OpTeX [1] is a new format prepared for LuaTeX. It keeps Plain TeX's simplicity, adds the power of the OPmac macros [2] and leaves behind the old obscurities with non-Unicode fonts and with various TeX engines. It provides a powerful font selection system, colors, graphics, references, hyperlinks, syntax highlighting, preparing indexes and bibliography listings. All these features are implemented at TeX macro level and they are ready to use without any external program. OpTeX is a new Plain TeX suitable for the present.

OpTeX was introduced in February 2020 and uploaded to CTAN. Now, it is ready to use in both TeX Live and MiKTeX with the basic command `optex document`. It underwent significant development in the first half of 2020, so please take the most recent version of it (from CTAN or a distribution) if you want to experiment with it.

### First example

A question was given on TeX.StackExchange [3]: “How can I write the symbol  $\text{t}$  in TeX?” Unfortunately, the accepted answer is typical for the old days of TeX: use a macro package which loads a font in an obscure 8-bit encoding and use the command `\textsubwedge{t}`. For me, the question has no sense and the answer sounds like something from the last millennium. Imagine a little modification of this question: “How can I write the symbol  $K$  in TeX?” And the answer should be: use package XY, then you can use `\printthischarK` command. But the normal answer is: “Use  $K$  in your text.”

My answer, second on that StackExchange page, was: “Use OpTeX and then use normally the symbol  $\text{t}$ . If a Unicode font supporting this character is loaded\* then there is no more problem.” For example:

```
\fontfam[Linux Libertine]
Symbol t
\bye
```

There was an addendum to my answer, essentially this: You can define `\def\t{\text{t}}` if your text editor or keyboard does not comfortably support making

---

\* More precisely: it is not a single character in this case but this is only a technical detail.

the  $\text{t}$ . Of course, TeX supports the command `\def` for such situations (among many others).

Unfortunately, there are many similar “problems”; you can see them at StackExchange or elsewhere. These “problems” shouldn't exist if we leave the old way of thinking about TeX. Today, there are plenty of good Unicode fonts. Simply, use them.

The second typical matter can be found in the accepted answer of that same StackExchange thread: “If you are using a special TeX engine then you must do `\ifsomething ... \fi` and you must load a package XY supporting this `\ifsomething`.” This is absurd. OpTeX eliminates the need to deal with such issues. It supports only one modern TeX engine: LuaTeX. Simplicity is power.

The three-line source file from the previous example shows another very important characteristic of Plain TeX. You need not *code* your document,\*\* you don't need to adapt yourself to a special computer language for your source files where are many `\begin{foo} ... \end{foo}`, for example. Simply *write* what you want. Of course, you have to decide what font family is used (the `\fontfam` command) and then just write the text. And use `\bye` if you want to say goodbye to TeX.

### Main principles of OpTeX

There are three main principles.

- The first one was mentioned in the previous paragraph. The author can *write* the document, he or she need not *code* the document in a computer language. The source text of the OpTeX document keeps the basic rules about tagging documents (chapters, sections, emphasized text, footnotes, listings of items, etc.), but there are minimal tagging marks, because we want to keep the text human-readable. There is a minimum of braces `{...}`, for example.
- The second principle of OpTeX says: don't hide TeX. Don't declare new parameters and new syntax constructions. If a user needs or wants to use TeX then he or she simply uses TeX. For example, we have the `\hsize` primitive register, and we don't declare a new one like `\textwidth`. If you need to set a value to the register then use simply `\hsize=15cm;` there is no alternative syntax like `\setlength{\textwidth}{15cm}`. Basic knowledge of the TeX primitive syntax is expected when using OpTeX. But it pays off.

---

\*\* Many questions at StackExchange begin with “My *code* is ...”.

- L<sup>A</sup>T<sub>E</sub>X, ConT<sub>E</sub>Xt and their many packages give users plenty of new parameters and options, and they create a new level of language (on top of T<sub>E</sub>X). When we are using or developing OpT<sub>E</sub>X, we don't need to go that same way. There is T<sub>E</sub>X for controlling the document, without new options and parameters. The macros of OpT<sub>E</sub>X are more straightforward and simple because they do not create a new level of syntax but only what is explicitly needed. If you need to make a change in the design of the document (for example) then you can copy the appropriate macro from OpT<sub>E</sub>X to your macro file and make the changes directly there. For example, there are macros `\_printchap`, `\_printsec` in OpT<sub>E</sub>X. Do you need a different design? Copy such macros to your macro file and declare your design there. This is the third principle of OpT<sub>E</sub>X which establishes a significant difference from L<sup>A</sup>T<sub>E</sub>X or ConT<sub>E</sub>Xt and which keeps the macros simple.

### Summary of features provided by OpT<sub>E</sub>X

The user manual of OpT<sub>E</sub>X is 21 pages. It contains hundreds of hyperlinks to a second part of the manual: technical documentation (about 140 pages). The technical documentation is generated directly from the OpT<sub>E</sub>X sources. There are listings of all OpT<sub>E</sub>X codes with extensive technical notes about the code.

We introduce a few features of the OpT<sub>E</sub>X system here, giving just short overviews.

**The font selection system.** You can use basic *variant selectors* `\rm`, `\bf`, `\it` and `\bi` as in Plain T<sub>E</sub>X, i.e. `\{bf text}`. Plain T<sub>E</sub>X does not define the `\bi` selector for bold italic, but OpT<sub>E</sub>X does because almost all font families used today provide this font variant.

You can choose the font family by the `\fontfam` command. WYSIWYG systems typically offer a menu for selecting the font family and this menu shows how text looks in listed fonts. This is a great advantage of such systems. You can get similar information by writing `\fontfam[catalog]`. Then all font families registered with the OpT<sub>E</sub>X macros are printed like a font catalog. Each font family is shown in all provided variants and the font modifiers given for each family are listed too. This “almost instantaneous” font catalog provides a sort of substitute for the interactive menus used in WYSIWYG systems.

Many font families provide *font modifiers*, for example `\cond` for condensed variants or `\caps` for capitals and small capitals. Usage of such font modifiers change a *font context*, but does not select the new font directly. This is done only when a variant selector is used. The variant selector respects the font context given by previous font modifiers. For example `\cond\it` selects condensed italics and if someone uses `\bf` in the same T<sub>E</sub>X group scope where `\cond` was declared then the bold condensed variant is selected.

There are many font modifiers declared among the font families. The set of available font modifiers depends on the selected font family. These modifiers can be independent of each other if the font family provides all such shapes. For example, `\cond` and `\caps` are independent, so you can set four font contexts by these two selectors and you can use four basic variant selectors: this gives 16 font shapes.

The settings of Unicode font features are implemented as font modifiers. This means that the current setting of font features is a part of the font context.

The setting of the font size is implemented as another font modifier. It means that the font size is the part of the font context too. If the family provides optical sizes, these sizes are respected by the OpT<sub>E</sub>X font selection system.

The font families are declared and registered by the font selection system in *font family files*. The family-dependent font modifiers are declared here. You can load more font families (by more `\fontfam` commands) and you can select between them by the *family selectors* like `\Termes`, `\Heros`, `\LMfonts`, etc. The font modifiers and variant selectors behave independently in each family, and they respect the selected family. For example if you want to mix Heros with Termes, you can declare:

```
\fontfam[Heros]
\fontfam[Termes] % Termes is current
\def\exhcorr{\setfontsize{mag.88}}
\famvardef\ss {\Heros\exhcorr\rm}
\famvardef\ssi{\Heros\exhcorr\it}
```

```
Compare ex-height of Termes
\ss with Heros \rm and Termes again.
```

This example shows several things:

- If multiple font families are loaded then the last one is selected as the current family.
- More variant selectors can be defined by the `\famvardef` command. The example shows a declaration of new `\ss` (sans serif) and `\ssi` (sans serif italic) variant selectors.

- The font size can be set by the `\setfontsize` command. It provides more syntactic rules but one of them is the keyword `mag`: the new font size is calculated as a factor of the font size currently selected.
- The Termes and Heros families have visually incompatible x-heights. We need to do the correction `Termes = 0.88 Heros`.

Macro programmers can declare *font selectors* directly with the `\font` primitive if the font name or font file name and its font features is known. Or the font selector can be declared by the `\fontlet\new=\ori <sizespec>` if another font selector is known, and we need only to set another font size of the same font. Finally, the font selector can be declared by the `\fontdef` macro if you can set the font by variant selector and font modifiers.

The last case (by the `\fontdef` macro) respects the actual font family and font context when the `\fontdef` macro is used. If you change the font family before a set of `\fontdef` declarations then all declarations are re-calculated to the new font family. Another example: you can set a new default font size by `\typo{11/13.5}` and all fonts declared by `\fontdef` will respect this new font size.

OpTeX loads only a few 8-bit Latin Modern fonts when its format is initialized. The Unicode fonts cannot be here due to a technical limitation of LuaTeX. It is supposed that these pre-loaded fonts will be used only for short experiments with OpTeX macros, not for processing real documents. A user should specify the font family with `\fontfam` first. This macro loads the Unicode variant of the fonts.

A lot of font families provided by OpTeX have registered the appropriate Unicode math font too. For example Latin Modern fonts have Latin Modern Math, Termes has TeX Gyre Termes Math, etc. The `\fontfam` macro loads this Unicode math font too unless the user says `\noloadmath`.

**Tagging the document.** All the typical tags for documents are borrowed from OPmac. The `op-demo.tex` document shows the basics of such tagging. Chapters are marked by `\chap{title}`, sections by `\sec{title}` and subsections by `\secc{title}`. The `{title}` ends at the end of the current line (unless the line ends with the % character; then the title continues). This decision mainly respects user needs: to *write* the document simply. Today, long lines (more than 80 characters) are quite common. Macro writers have a little complication if they use `\chap`, etc., in their macros because the end of line is changed locally only at the input processor level, but this can be handled.

Labels for cross-references can be declared by `\label[<label>]` or by `\sec[<label>] <title>`, etc. The labels can be used by the commands `\ref[<label>]` or `\pgref[<label>]`.

Lists of items look like:

```
\begitems
* First idea.
* Second idea.
\begitems \style i
* First subidea.
* Second subidea.
\enditems
\enditems
```

**Auto-generated listings.** The table of contents can be printed by the `\maketoc` command and the index by the `\makeindex` command. The alphabetical sorting of the index is done at the TeX macro level with respect to the rules of the current language selected. No external software is needed. Thus, you don't need to do more than specify the words to be indexed and write `\makeindex`. The index is reinitialized in each TeX run.

The situation is similar with bibliographies. OpTeX reads `.bib` database files directly without the need of any external program and creates these listings with respect to a big set of rules declared in the *bib-style files*. These rules and customizing possibilities are described in [4]. OpTeX provides `simple` and `iso690` bib-style files now.

**Colors.** Colors can be simply used, for example, `\Red something` or `\Magenta text`. They can be defined in three possible ways:

```
\def \Red {\setrgbcolor {1 0 0}}
\def \Magenta {\setmykcolor {0 1 0 0}}
\def \Brown {\setmykcolor {0 0.67 0.67 0.5}}
\def \Black {\setgreycolor {0}}
```

and the user can define more such colors. The respective color model (RGB or CMYK) is used in low-level PDF commands. If you need not mix color models in the PDF output then you can say `\onlyrgb` or `\onlycmyk`. Then the colors are recalculated to the desired color model as needed. This calculation is done only via simple math formulae; the visual feeling of the color may be changed. These two color models are not transformable from one to the other without loss of information.

OpTeX initializes two color stacks: one for normal text and a second for footnotes. Footnotes can span from one page to another independently on the main text, so we need two independent color stacks. You can create a long footnote in green, for example, with the main text in red at the same point.

The next page continues with the red main text and green footnote. All colors work without problems on the next page. (If you try to do the same in L<sup>A</sup>T<sub>E</sub>X, you realize that it does not work without special care.)

The color blender macro `\colordef` is provided by OpT<sub>E</sub>X. It enables color mixing in the subtractive (CMYK) or additive (RGB) color model.

**Graphics.** The `\inspic {<file name>}` includes a graphics file in JPEG or PNG or PDF format (the last can be a vector graphic) at the current typesetting point as an `\hbox`. The width or height of the picture can be given by `\picwidth` or `\picheight` parameters. Other parameters accepted by the `\pdfximage` primitive can be specified too. For example, you can select a given page from the included PDF file.

Inkscape (a free vector graphics editor) is able to save a vector graphic to a PDF file and labels to a L<sup>A</sup>T<sub>E</sub>X file. OpT<sub>E</sub>X is able to read both these files (the L<sup>A</sup>T<sub>E</sub>X commands used by Inkscape must be emulated here). You can do this by `\inkinspic` macro which outputs the PDF graphic plus the labels. They are printed in the current fonts selected in the document.

OpT<sub>E</sub>X supports linear transformations using commands `\pdfrotate`, `\pdfscale` and (in general) `\pdfsetmatrix`. All compositions of these operations are allowed too. The `\transformbox` macro does linear transformations and the real boundaries of the box are calculated in respect of the transformed material.

If the graphics need to interact with the text, then TikZ can be used (`\input tikz`). This works in Plain T<sub>E</sub>X too. But simple tasks can be done using OpT<sub>E</sub>X macros without TikZ (we are happy when TikZ is not loaded because TikZ is a very big package). For example, putting the text into an oval or into an ellipse (its size depends on the amount of the text) can be done directly by `\inoval` or `\incircle` OpT<sub>E</sub>X macros. A clipping path can be declared by `\clipinoval` or `\clipincircle`.

**Hyperlinks.** There are four types of internal links: cross-references, citations (bibliography), links from the table of contents or index, and hyperlinks to/from footnotes. There is one type of external link generated by `\url` or `\ulink` macros. The hyperlinks can be activated by the `\hyperlinks` or `\fnotelinks` commands. The user or macro programmer can declare more types of hyperlinks.

Structured outlines (for PDF viewers) are automatically generated by the `\outlines` macro.

**Verbatim text.** Code listings can be placed between a `\begtt` and `\endtt` pair, or they can be included from an external file with, e.g., `\verbinput (<fromline>-<toline>) filename.c`. Inline verbatim text can be surrounded by an arbitrary character declared by the `\activettchar` macro. Nowadays, the most common usage is `\activettchar`` as a declaration. Then you can type '`\relax`' to print `\relax`. This tagging is inspired by the Markdown language and is used very commonly at StackExchange, for example.

Sometimes you need to use inline verbatim in titles or parameters of other macros. This doesn't work when the `\activettchar` character is used because there is a "catcode movement" in the parameter of '...'. OpT<sub>E</sub>X provides a robust alternative command for such situations: `\code{<text>}`. The `<text>` is printed detokenized with `\escapechar` set to -1. From the user point of view, all "sensitive" characters in the `\code` parameter `<text>` should be escaped. For example, `\code{\relax\{}` prints `\relax{}`. This can be used in titles of sections, etc., without problems.

Listings can be printed with highlighted syntax (typically colored). Such syntax highlighting is defined in *hisyntax* macro files and can be activated with `\begtt \hisyntax{C} ... \endtt`, for example. All processing is done at the T<sub>E</sub>X macro level without using any external programs. These *hisyntax* files are easily customizable. They support C, XML/HTML, T<sub>E</sub>X and Python syntax at this time; others may be added in the future. Users can declare more such files.

**Languages.** LuaT<sub>E</sub>X is the only T<sub>E</sub>X engine which enables loading hyphenation patterns for a selected language on demand inside the document. Thus, we need not preload all hyphenation patterns in the format. Hooray! OpT<sub>E</sub>X provides the language selectors `\(isocode)lang` (for example `\enlang`, `\frlang`, `\delang`, `\eslang`, `\cslang`). These commands load the hyphenation patterns of a given language when they are first used in the document, and switch to the loaded hyphenation patterns when they are used subsequently. Macro programmers can set more language-dependent macros; these macros are processed when an `\(isocode)lang` language selector is used.

Language-dependent phrases like "Chapter", "Figure", "Table" are automatically selected by the current value of the `\language` primitive register (this is used for hyphenation patterns). These phrases are declared in OpT<sub>E</sub>X via:

\_langw	en	Chapter	Table	Figure	Subject
\_langw	cs	Kapitola	Tabulka	Obrázek	Věc
\_langw	de	Kapitel	Tabelle	Abbildung	Betreff
\_langw	es	Capítulo	Tabla	Figura	Sujeto

Quotation mark pairs can be declared by `\quoteschars{clqq}{crqq}{clq}{crq}`, for example `\quoteschars{"'"}'` for English. The first type of quotation marks can be printed by `"text"` and the second type by `\'text'.`\* Several languages have their `\quoteschars` predefined in OpTeX.

**Styles in OpTeX.** The default design style of the document is inspired by Plain TeX: 10pt/12pt size of basic text, 20pt `\parindent`, zero `\parskip`.

The command `\report` at the beginning of the document sets some typesetting parameters differently, suitable for reports. The `\letter` command sets a design convenient for letters.

If you write `\slides` then you can create presentation slides. This style is documented in the file `op-slides.tex` which also serves as an example of usage of this style.

**Name spaces for control sequences.** Suppose that the user writes `\def\fi{Finito}` into the document. What happens? When LATEX, ConTEXt or original Plain TeX is used then the document processing crashes. When OpTeX is used, then nothing critical happens. The user name space of control sequences allows names where only letters are used. If such sequences are redefined by users then this only affects their own usage and macros; it's not a problem for the internal macros of OpTeX. The internal macros of OpTeX do not use such control sequences.\*\*

When OpTeX initializes, all TeX primitives and OpTeX macros have two representations, prefixed: `\_hbox` and unprefixed: `\hbox`. OpTeX uses only the prefixed versions. This is the OpTeX name space. A user can work with the non-prefixed versions of control sequences. If he or she redefines them nothing happens with the OpTeX internal macros.

---

\* When `\quoteschars` are declared, then the original Plain TeX macros `"` and `\'` are redefined. This problem is discussed further in the following section about compatibility with Plain TeX.

\*\* There is only one exception: the control sequence `\par` is (unfortunately) hardwired to the TeX internal algorithms—it is the output of the tokenizer when an empty line occurs in the input. If the user redefines `\par` by mistake then processing may crash.

The internal OpTeX macros not intended for direct usage by the user have only a prefixed form. And the control sequences never used in the OpTeX macros but offered to the user (`\alpha` and other sequences for math symbols) are defined only in unprefixed form (in the user name space).

The character `_` always has category code 11 (letter) in OpTeX. You aren't forced to write `\makeatletter` or anything similar when you need to access the control sequences from the OpTeX name space. Simply use them. You can redefine these control sequences, but then presumably you know what you are doing. An example of when it's expected to redefine macros from the OpTeX name space was given above where the macros `\_printchap` and `\_printsec` were mentioned.

The character `_` has category code 11 in math mode too. It is defined as math-active for doing subscripts in math formulae. Cases like `\int_a^b` work too because they are handled in the LuaTeX input processor.

OpTeX uses the `_` character only as the first character of control sequences. We suppose that macro package writers will use internal control sequences in the form `\_pkg_foo`. This is a package name space. Moreover, the macro writer does not need to see repeated `\_pkg_foo`, `\_pkg_bar`, `\_pkg_other` control sequences in the code because there is the command `\_namespace{pkg}`. When it is used then the macro writer can use `\.foo`, `\.bar`, `\.other` in the code which is much more human-readable. These control sequences are converted to internal `\_pkg_foo`, etc., automatically by the LuaTeX input processor.

**Odds and ends.** Logos are defined with an optional / character which can follow the control sequence; it is ignored if present. You can write, for example:

```
\OpTeX/ is a new generation of Plain \TeX/
with features comparable to \LaTeX.
But \LaTeX/ needs to load about ten additional
packages to have comparable features.
```

This source looks more attractive. We needn't separate such control sequences by `{}` or some similar construction.

The command `\lorem[<from>-<to>]` produces the text “Lorem ipsum dolor sit”. There is an interesting implementation of this macro: the 150 paragraphs of the text are not loaded into the OpTeX format. Rather, the first usage of the `\lorem` command loads the external file `lipsum.ltd.tex` from

the L<sup>A</sup>T<sub>E</sub>X package `lipsum` and prints the given paragraphs. The second and subsequent usage of the `\lorem` macro prints the desired text from memory.

**Compatibility with Plain T<sub>E</sub>X.** All Plain T<sub>E</sub>X macros were re-implemented in OpT<sub>E</sub>X; nearly all features of Plain T<sub>E</sub>X are essentially preserved. But there are some differences.

- The internal control sequences like `\p@` or `\f@c@t` were renamed or completely removed. We don't support the "catcode dancing" with the @ character.
- The Latin Modern 8-bit fonts in the EC encoding are preloaded instead of the Computer Modern 7-bit fonts.
- The math fonts are preloaded in 7-bit versions comparable to Plain T<sub>E</sub>X plus AMST<sub>E</sub>X. But if the `\fontfam` command is used then the preloaded 8-bit text fonts and 7-bit math fonts are not used; instead, Unicode text and Unicode math fonts are used.
- The control sequences for characters defined by `\mathhexbox` or in a similarly obscure way are not defined (for example `\P`, `\L`). We suppose that such characters should be used directly in Unicode. Only the `\copyright` macro is kept but it is defined by `\def\copyright{©}`.
- The accent macros `\", \', \v, \u, \=, \^, \., \H, \~, \``, `\t` are undefined in OpT<sub>E</sub>X. We are using Unicode, so all accented characters can be written directly and this is the only recommended way. You can use these control sequences for your own purposes. For example `\"` and `\'` are used for quotation marks when `\quoteschars` are declared, as mentioned earlier. If you insist on using old accents from Plain T<sub>E</sub>X then you can use the `\oldaccents` command.
- The default paper size is A4 with 2.5 cm margins, not letter with 1 in margins. You can declare the default Plain T<sub>E</sub>X margins by the command `\margins/1 letter (1,1,1,1)in.`
- The page origin is at the top left page corner, not at the coordinates [1in, 1in] as in Plain T<sub>E</sub>X. This is a much more natural setting. These "1 in" values brought only unnecessary complications for macro programmers.
- The `\sec` macro is reserved for sections, not the math secant operator.

**Tips and tricks.** The web page [5] has a section Tips and Tricks when using OpT<sub>E</sub>X. This is inspired by Tips and Tricks of OPmac [6]. OpT<sub>E</sub>X users can give a problem and I'll try to put the solution here.

## My path from T<sub>E</sub>X to OpT<sub>E</sub>X

My first attempts at T<sub>E</sub>X were with Plain T<sub>E</sub>X in 1991. I realized that it was unusable with the Czech language until 8-bit support of fonts was ready. The concept of writing `Ol\v s\ak` instead of directly `Olšák` is not viable for real Czech texts. And Czech hyphenation patterns cannot work in the former case either.

I became a member of the development team of C<sub>S</sub>T<sub>E</sub>X. The 8-bit fonts with Czech and Slovak alphabet (CSfonts) were created using METAFONT, derived from the Computer Modern fonts. I created a macro to read the `plain.tex` file without the part of loading Computer Modern fonts. This part was replaced by loading CSfonts. The Czech and Slovak hyphenation patterns were added and the CSplain format was originated. I was (and still am) a maintainer of CSplain. In the mid-1990s, I added PostScript support to CSfonts and Czech and Slovak accents for the 35 base Adobe fonts.

I read *The T<sub>E</sub>Xbook* and felt that the description of all T<sub>E</sub>X algorithms could be done more systematically and clearly. This was the reason why I wrote the *T<sub>E</sub>Xbook naruby* ("T<sub>E</sub>Xbook inside out", in Czech only) [7]. This book became a standard for T<sub>E</sub>X manuals in Czech. For example, computer science students were using it to help welcome their new colleagues.

L<sup>A</sup>T<sub>E</sub>X was not the center of my interest because it seems to be more complicated. I wrote an article "Why I don't like using L<sup>A</sup>T<sub>E</sub>X" (1997, in Czech) [8]. I have reread this article recently and I found that all its arguments are still valid. The main problem is that L<sup>A</sup>T<sub>E</sub>X offers a "coding language" not a "human language" for writing documents. The second problem is that it tries to hide T<sub>E</sub>X by creating a new level of language. But this is impossible because T<sub>E</sub>X was not designed for such a task. L<sup>A</sup>T<sub>E</sub>X users will still see the T<sub>E</sub>X messages like "extra alignment tab has been changed to `\cr`". This language is different from the language used in L<sup>A</sup>T<sub>E</sub>X manuals, so L<sup>A</sup>T<sub>E</sub>X users are lost. From my point of view, it is unfair to T<sub>E</sub>X users to hide T<sub>E</sub>X.

I created encT<sub>E</sub>X in 2003 [9]. It is a pdfT<sub>E</sub>X extension which supports input of UTF-8 encoded documents directly. The Unicode characters (represented by multi-byte sequences in UTF-8) are mapped by encT<sub>E</sub>X to one 8-bit character or to a control sequence which can be defined arbitrarily. The most important advantage is that each Unicode character is represented as only one token in T<sub>E</sub>X. This is the main difference from the L<sup>A</sup>T<sub>E</sub>X package `\inputenc`.

I have been giving lessons about  $\text{\TeX}$  to our students and learning from them. They represent a new generation and their point of view and requirements to  $\text{\TeX}$  are very important to me. One of the results of these discussions is: at present, only Unicode makes sense. I decided that enc $\text{\TeX}$  development was a dead end. We have a sufficient number of quality Unicode fonts today, we don't need to do special mappings and complicated macros, we can use Unicode fonts directly.

As a maintainer of CSplain and (of course) user of Plain  $\text{\TeX}$ , I created many typical macros needed for document processing: creating the table of contents, fonts in different sizes, references, hyperlinks, etc. I released these home-made macros in 2013 as an additional package in CSplain called OPmac [2]. It works with CSplain or Plain  $\text{\TeX}$  with all typical  $\text{\TeX}$  engines. The great disadvantage of OPmac is that its technical documentation, though extensive, is only in the Czech language, because it was created as home-made documentation of home-made macros.

I created the template for student theses at our university based on OPmac and CSplain [10]. There are hundreds of satisfied users. It shows that Plain  $\text{\TeX}$  is still viable today.

I planned to make a re-implementation of OPmac with new English documentation and with new features and internals. What  $\text{\TeX}$  engine would be suitable for such a plan? X $\text{\TeX}$  does not support all of the micro-typographic extensions introduced by pdf $\text{\TeX}$ , and does not offer the extensive customization of Lua $\text{\TeX}$ . Furthermore, it seems that X $\text{\TeX}$  is no longer significantly developed, while Lua $\text{\TeX}$  has been declared substantively stable as of version 1.10 (<http://www.luatex.org/roadmap.html>). Lua $\text{\TeX}$  won.

I finished my reimplementations of OPmac by May 2020 and the result is called Op $\text{\TeX}$ .\* The documentation of Op $\text{\TeX}$  expects knowledge of  $\text{\TeX}$  basics. This is the main reason why I wrote a short encyclopedic document “ $\text{\TeX}$  in a Nutshell” [11] (in English). The big reference “ $\text{\TeX}$ book naruby” [7] and short summary “ $\text{\TeX}$  pro pragmatiky” [12] are available only in the Czech language, unfortunately.

I hope that Op $\text{\TeX}$  will find many users and thus gain more respect. I hope that it will be a good alternative to other currently used formats. It can show that the native principles of  $\text{\TeX}$  do not have to be covered by new levels of computer languages, and they can live at present: 40 years after the birth

of  $\text{\TeX}$ . My dream is to eliminate the widespread notion among  $\text{\TeX}$  users that  $\text{\TeX}$  is equal to L $\text{\TeX}$ . Will you help me with it?

## References

1. <http://petr.olsak.net/optex>
2. P. Olšák: OPmac: Macros for Plain  $\text{\TeX}$ . *TUGboat* 34:1, 2013, pp. 88–96. [petr.olsak.net/opmac-e.html](http://petr.olsak.net/opmac-e.html) [tug.org/TUGboat/tb34-1/tb106olsak-opmac.pdf](http://tug.org/TUGboat/tb34-1/tb106olsak-opmac.pdf)
3. [tex.stackexchange.com/questions/541064](https://tex.stackexchange.com/questions/541064)
4. P. Olšák: OPmac-bib: Citations using \*.bib files with no external program. *TUGboat* 37:1, 2016, pp. 71–78. [tug.org/TUGboat/tb37-1/tb115olsak-bib.pdf](http://tug.org/TUGboat/tb37-1/tb115olsak-bib.pdf)
5. [petr.olsak.net/optex/optex-tricks.html](http://petr.olsak.net/optex/optex-tricks.html)
6. [petr.olsak.net/opmac-tricks-e.html](http://petr.olsak.net/opmac-tricks-e.html)
7. P. Olšák:  *$\text{\TeX}$ book naruby*, 1996, 2000. 468 pp., ISBN 80-7302-007-6. Freely available. [petr.olsak.net/tbn.html](http://petr.olsak.net/tbn.html)
8. P. Olšák: *Proč nerad používám L $\text{\TeX}$* , 1997. [petr.olsak.net/ftp/olsak/bulletin/nolatex.pdf](http://petr.olsak.net/ftp/olsak/bulletin/nolatex.pdf)
9. [petr.olsak.net/enctex.html](http://petr.olsak.net/enctex.html) (2003–)
10. P. Olšák: The CTUstyle template for student theses. *TUGboat* 36:2, 2015, pp. 130–132. [petr.olsak.net/ctustyle.html](http://petr.olsak.net/ctustyle.html) [tug.org/TUGboat/tb36-2/tb113olsak.pdf](http://tug.org/TUGboat/tb36-2/tb113olsak.pdf)
11. [ctan.org/pkg/tex-nutshell](http://ctan.org/pkg/tex-nutshell) (2020–)
12. P. Olšák:  *$\text{\TeX}$  pro pragmatiky*, 2013, 2016. 148 pp, ISBN 978-80-901950-1-1. Freely available. [petr.olsak.net/tpp.html](http://petr.olsak.net/tpp.html)

◇ Petr Olšák  
Czech Technical University  
in Prague  
<http://petr.olsak.net>

---

\* You can guess why I had more time to do it in the year 2020.

Book reviews: *Robert Granjon, letter-cutter*, and *Granjon's Flowers*, by Hendrik D.L. Vervliet

Charles Bigelow

Hendrik D.L. Vervliet, *Robert Granjon, letter-cutter; 1513–1590: an oeuvre catalogue*.

Oak Knoll Press, New Castle, DE, 2018, hc, 200 pp., US \$75.00, ISBN 978-1584563761, [oakknoll.com/pages/books/131957](http://oakknoll.com/pages/books/131957).

Hendrik D.L. Vervliet, *Granjon's Flowers: An Enquiry Into Granjon's, Giolito's, and De Tournes' Ornaments, 1542–1586*.

Oak Knoll Press, New Castle, DE, 2016, hc, 248 pp., US \$65.00, ISBN 978-1584563556, [oakknoll.com/pages/books/127576](http://oakknoll.com/pages/books/127576).



These two books tell us essentially all that we know of the illustrious sixteenth century French type designer Robert Granjon. Thanks to six decades of meticulous research by Hendrik D. L. Vervliet, Granjon can now be ranked not only among the finest letter-cutters<sup>1</sup> of the French Renaissance but among the greatest of all time. Granjon was both astonishingly versatile and amazingly productive, cutting around ninety fonts, including romans, italics, and cursive blackletters in the Latin alphabet, as well as Greek, Cyrillic, Arabic, Armenian, Hebrew, and Syriac among non-Latin scripts. He also cut music fonts and ornamental printers' flowers. Vervliet quotes Giambattista Bodoni, writing two centuries after Granjon's death, that: "Maestro Robert Granjon was the best letter-cutter ever." Bodoni, a

<sup>1</sup> Granjon would today be called a "type designer", but that term was not used until the twentieth century. Vervliet instead uses "letter-cutter" signifying the material technology of type employed from the fifteenth through the nineteenth century, when types were created by skilled artists who used hard steel engraving tools and files to cut and shape the forms of letters at actual size in steel punches. The punches were struck into copper matrices, producing concave impressions of the letters. From the matrices, types were cast in lead alloy, with the letter forms in relief, to be inked and impressed onto paper in printing. A common term for those type artists has been "punch-cutter", which denotes the technique but omits the art. A sculptor analogously described might be called a "marble cutter" or "clay shaper", and a painter, a "color dauber" or "pigment brusher".

preeminent type artist himself, prized regularity, clarity, good taste, and grace in type design and evidently saw those qualities in Granjon's work.

The sixteenth century has been called the Golden Age of Typography; the most famous letter-cutter of the era was Claude Garamond, renowned for his roman types. More than fifty digital type families today are named Garamond or Garamont, though many are not based on Garamond's actual designs.

Granjon's name, though well known in his time, is borne today by only one type family, which combines a roman based on a Garamond design with an italic based on a Granjon design. The only type family today that authentically, and brilliantly, embodies both Granjon's roman and his italic designs is not named "Granjon" but "Galliard", the English spelling of "Gaillard", the original name of a small (around 8 point) roman that Granjon cut around 1570. (This review is typeset in Galliard, both roman and italic.) Before and during Granjon's early career, italic was often used in body text instead of roman, but later, italic was used in subordination to roman—usually a Garamond roman with a Granjon italic. By the seventeenth century, italic was commonly paired with roman in type families, a practice continuing to present. Hence, though Granjon's italics have often been imitated in Garamond type revivals, they are typically known as "Garamond Italic".

Perhaps paradoxically, the subordination of italic to roman gave Granjon greater freedom of expression and invention in cutting italics, which could be more exuberant and stylish because they did not need to support fluent reading of long texts. Granjon cut some thirty different italics in at least three different styles. His "pendante" or "couchée" is a strongly slanted style that has most often been revived as a companion to Garamond romans. His "droite" is a narrower and more upright style, revived and modified by Jan Tschichold for Sabon, a modern revival of Garamond roman and Granjon italic, produced for Monotype and Linotype hot-metal composition machines and then adapted to photo- and digital type. A third italic style of Granjon resembled models of Italian chancery cursive like those of writing manuals by Arrighi and Tagliente. This chancery style is the italic in the Galliard family.

In addition to plentiful details, dates, and examples of Granjon's typefaces, Vervliet provides a biography of the man. Robert Granjon was born around 1513 in Paris, son of a bookbinder and publisher. He was apprenticed to a goldsmith and afterward turned to letter cutting. The earliest types reliably attributed to him are a Roman on "Long Primer" body ( $\approx$  10 point today) in 1542 and an Italic on "English" body ( $\approx$  14 point today). Though working in Paris from

around 1542 to 1552, Granjon traveled to Lyons and sold types to printers there, especially Jean de Tournes, for whom he also cut some exclusive types. Around 1553, Granjon moved to Lyons and later married Antoinette Salomon, daughter of Bernard Salomon, an eminent illustrator, engraver, and book designer for De Tournes. He apparently left Lyons around 1562, worked for a time in Geneva and Strasbourg, and in 1564 moved to Antwerp, where he sold types and custom cuttings to Christophe Plantin, known as the king of printers. Granjon moved back to Paris in 1570, to Lyons from 1575 to 1578, and then to Rome, where he cut punches for some two dozen non-Latin types and printers' flowers for the Vatican, until a few months before his death in 1590.

The types of Granjon became internationally popular when French romans and italics, betokening fashionable Renaissance humanism, became popular not only in France, but also in Italy, Switzerland, Spain, and the Low Countries. In his first ten years of letter-cutting in Paris, Granjon cut around 24 fonts: 11 romans, 9 italics, 2 Greeks, and 2 musics, and 12 fleurons (typographic "flowers" or ornaments). These amount to some 2,500 punches, averaging 250 punches per year, an impressive number considering that he also cast ("founded") some of his types and was also a publisher, producing around 14 books, including works of religion, science, history, music, and literature. He spent nine years in Lyon, cutting around 2600 punches, including 10 fonts of italic, 4 Greeks, 2 Civilités (a cursive French blackletter), 4 musics, 12 sets of fleurons, and two sets of large script initials (the models cut in wood and cast in sand).

By the time of his death, Granjon's italics dominated European humanist (non-blackletter) typography. In Christophe Plantin's 1585 Folio Specimen, twelve of thirteen italics are by Granjon. In the 1592 Egenolff-Berner type foundry specimen produced in France, seven of the eight italics are by Granjon.

There is no known portrait nor physical description of Granjon the man, but his career seems all the more impressive and romantic when we consider his travels. As a free-lance letter-cutter, Granjon presumably rode on horseback from city to city, carrying his gravers and files in a box, along roads sometimes rough and hazardous. The gunslinger Paladin in a 1950s American television show had a calling card that read, "Have Gun Will Travel". Granjon's card could have read, "Have Graver Will Travel".

### *Granjon's Flowers*

In addition to alphabetic and music types, Granjon cut ornamental printers flowers or "fleurons". Ornamental flowers, presumably inspired by Islamic patterns or

"arabesques" in metal work, bookbinding, and other crafts, had been used in European printing long before Granjon. In *Granjon's Flowers*, Vervliet provides a concise historical analysis of four stylistic phases of French ornamental typography in the sixteenth century, placing Granjon's work in the fourth phase. Beginning in the 1540s, Granjon was cutting single ornaments, and by the mid-1560s, he was creating complex sets of modular flowers that could be combined into a wide variety of patterns. As Vervliet puts it, these were "arabesque patterns based on a continuous repetition of non-figurative foliated elements and undulating, intertwined, schematized and denaturalized stems". The cutting of flowers gave Granjon a golden, or leaden, opportunity for abstract expression in typography, and he obviously reveled and excelled in it, cutting around fifty elegant, graceful, and original abstractions, which Vervliet has painstakingly tracked down, identified, and documented in an impressive display of typographic detective work.

This definitive study provides examples, dates, and notes of all the Granjon flowers that Vervliet has identified, accompanied by an extensive bibliography.

Several of Granjon's flower designs were revived by Monotype in the 1920s and have been known as "Granjon Arabesques". For decades, they have intrigued and delighted typographers who have explored myriad possible combinations by rearranging the elements. Most of these works were printed letterpress in limited-edition books now unobtainable or astronomically priced, but Jacques André has made a brilliant digital edition and translation of Swiss typographer Max Caflisch's playful explorations of Granjon's Arabesque, "Kleines Spiel mit Ornamenten" (in French translation: "Petits jeux avec des ornements"). It includes examples of the ornaments in use in historical books along with reproductions of Caflisch's studies, not by scanning digitization but by new compositions using digital fonts of Monotype's Granjon Arabesques. It is available at [jacques-andre.fr/ed/caflisch-jeux.pdf](http://jacques-andre.fr/ed/caflisch-jeux.pdf).

Both books were handsomely designed by Alastair Johnston, fine printer and typographic scholar. *Robert Granjon, letter-cutter* is composed in Stempel Garamond, which has a Granjon italic, and *Granjon's Flowers* in Linotype Granjon, with a Garamond roman and Granjon italic.

In closing, two references; and sample pages are reproduced following.

Bigelow, C. On Type: Galliard. *Fine Print* 5(1):27–30, 1979. Reprinted in *Fine Print on Type*, 1990.

Carter, M. Galliard: a modern revival of the types of Robert Granjon. *Visible Language* 19(1):77–97, 1985.

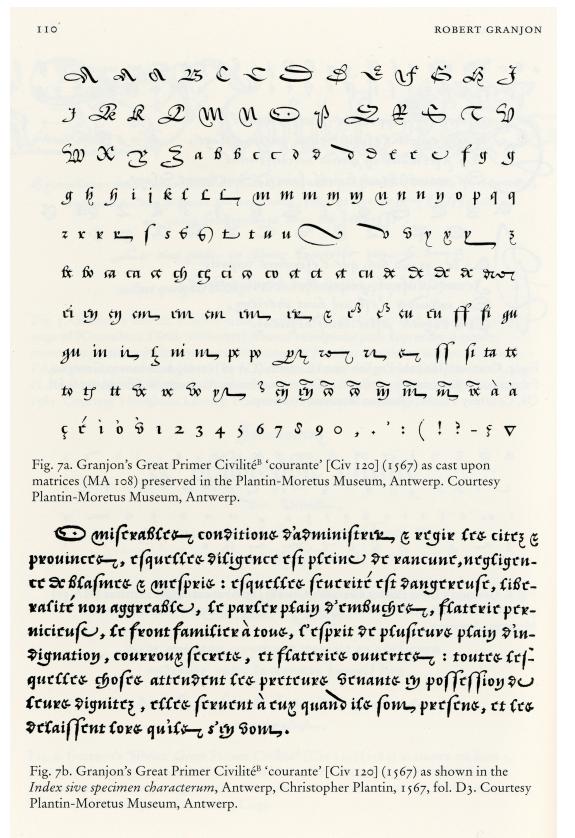


Fig. 7a. Granjon's Great Primer Civilité® 'courante' [Civ 120] (1567) as cast upon matrices (MA 108) preserved in the Plantin-Moretus Museum, Antwerp. Courtesy Plantin-Moretus Museum, Antwerp.

**C**onfiscables, conditions d'administre, et regir les citez & prouince, esquellese diligence est pleine de rancune, negligen-  
ce de blesme, & mespris : esquellese seurte est dangereuse, lise-  
ralité non aggregable, le parler plaiy d'embûches, flaterie per-  
niciouse, le front familiar à tous, l'esprit de plusiure plaiy d'in-  
signiatoy, courroux secrete, et flatterie ouverte : toutes ces  
esquellese chose attendent les perteure de nantez y possesyon de  
seure dignitez, elles serent à eux quand ilz sonz, pfecte, et les  
delassent lors qu'ilz s'y vont.

Fig. 7b. Granjon's Great Primer Civilité® 'courante' [Civ 120] (1567) as shown in the *Index sive specimen characterum*, Antwerp, Christopher Plantin, 1567, fol. D3. Courtesy Plantin-Moretus Museum, Antwerp.



Fig. 43a. Granjon's seven-piece combinable flower on Two-line Pica [FLC 8.5] (1571) as it appears in Euclides, *Elementa, libris XV ad germanam geometriæ intelligentiam... restituta*, Coloniae Ubiorum, s. n., 1602, fol. a1, p. 1; Plantin-Moretus Museum, Antwerp, B 446 [colophon: Lugduni: ex officina Joannis Tornæsi, 1578]; variant address: Lutetiae, apud Iacobum Du Puis, 1578; Cartier, 2: 584, no. 585.

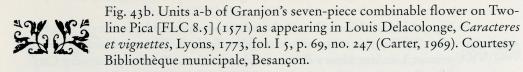


Fig. 43b. Units a-b of Granjon's seven-piece combinable flower on Two-line Pica [FLC 8.5] (1571) as appearing in Louis Delacolonge, *Caractères et vignettes*, Lyons, 1773, fol. I 5, p. 69, no. 247 (Carter, 1969). Courtesy Bibliothèque municipale, Besançon.

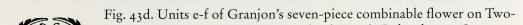


Fig. 43c. Units c-d of Granjon's seven-piece combinable flower on Two-line Pica [FLC 8.5] (1571) as appearing in Louis Delacolonge, *Caractères et vignettes*, Lyons, 1773, fol. I 6, p. 70, no. 248 (Carter, 1969). Courtesy Bibliothèque municipale, Besançon.

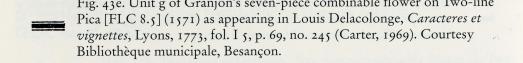


Fig. 43d. Units e-f of Granjon's seven-piece combinable flower on Two-line Pica [FLC 8.5] (1571) as appearing in Louis Delacolonge, *Caractères et vignettes*, Lyons, 1773, fol. I 6, p. 70, no. 249 (Carter, 1969). Courtesy Bibliothèque municipale, Besançon.

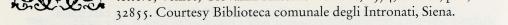


Fig. 43e. Unit g of Granjon's seven-piece combinable flower on Two-line Pica [FLC 8.5] (1571) as appearing in Louis Delacolonge, *Caractères et vignettes*, Lyons, 1773, fol. I 6, p. 70, no. 245 (Carter, 1969). Courtesy Bibliothèque municipale, Besançon.

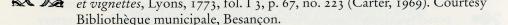


Fig. 44a. Granjon's six-piece combinable flower on Great Primer® [FLC 6] (1569) with variant units a-b as used in Bernardo Tasso, *Le lettere*, Venice, Giovanni Antonio Bertano, 1574, fol. A1, p. 1; CNCE 32855. Courtesy Biblioteca comunale degli Intronati, Siena.

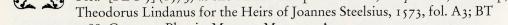


Fig. 44b. Variants of unit (a-b) of Granjon's six-piece combinable flower on Great Primer® [FLC 6] (1569) as shown in Delacolonge's *Caractères et vignettes*, Lyons, 1773, fol. I 3, p. 67, no. 223 (Carter, 1969). Courtesy Bibliothèque municipale, Besançon.



Fig. 45. Units a-b of Granjon's two-piece combinable flower on Small Pica® [FLC 5] (1573) as shown in Radulphus Ardens, *In epistolæ*, Antwerp, Theodorus Lindanus for the Heirs of Joannes Steelius, 1573, fol. A3; BT 4988. Courtesy Plantin-Moretus Museum, Antwerp.

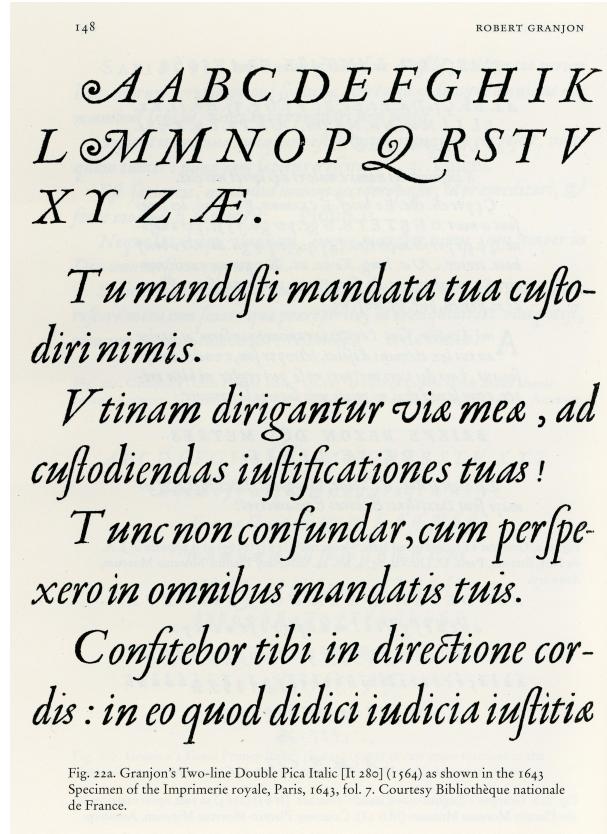


Fig. 22a. Granjon's Two-line Double Pica Italic [It 280] (1564) as shown in the 1643 Specimen of the Imprimerie royale, Paris, 1643, fol. 7. Courtesy Bibliothèque nationale de France.

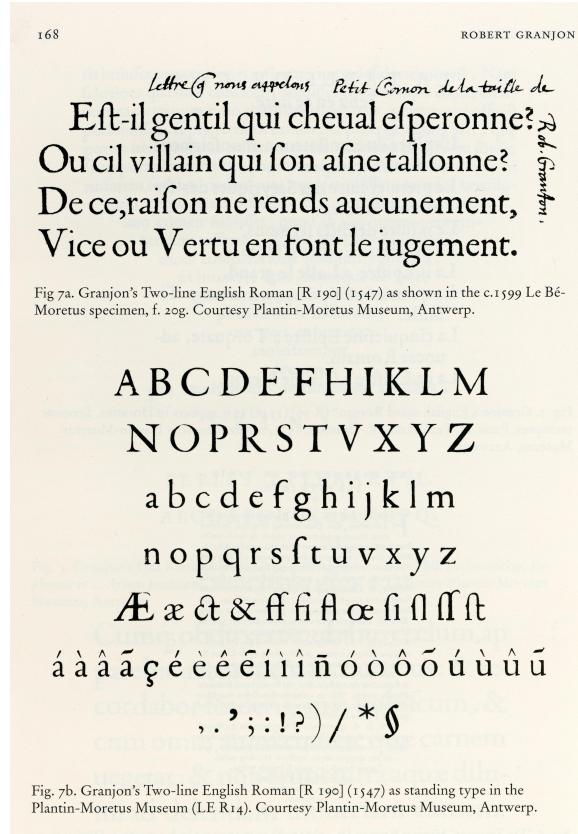


Fig. 7a. Granjon's Two-line English Roman [R 190] (1547) as standing type in the Plantin-Moretus Museum (LE R14). Courtesy Plantin-Moretus Museum, Antwerp.

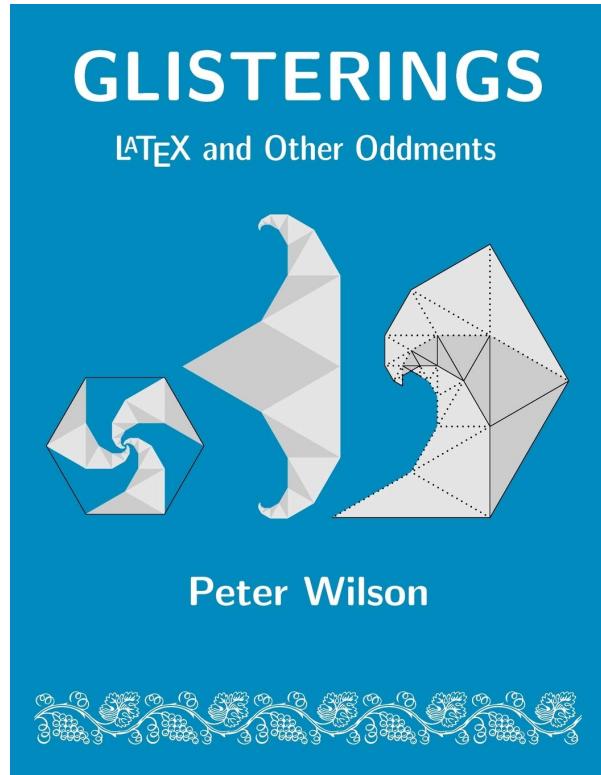
All pages reproduced from *Robert Granjon, letter-cutter*.

Book reviews: *Robert Granjon, letter-cutter*, and *Granjon's Flowers*, by Hendrik D.L. Vervliet

## Book review: *Glisterings*, by Peter Wilson

Boris Veytsman

Peter Wilson, *Glisterings. L<sup>A</sup>T<sub>E</sub>X and Other Oddments*. T<sub>E</sub>X Users Group, Portland, Oregon, USA, 2020, paperback, 130pp., US\$15.00, ISBN 978-0982462621.



Many many years before people were reading `tex.stackexchange.com` on their smartphones, the primary forums for asking T<sub>E</sub>X questions were the list `texhax@tug.org` (still in lively existence) and the Usenet group `comp.text.tex` (also still in existence, though not so lively), or `ctt` as it was known to aficionados. I guess that before `texhax` and `ctt`, in the prehistoric times when people exchanged T<sub>E</sub>X tapes and freshly knapped stone axes, there were T<sub>E</sub>X-related BBSes, but I do not have reliable information about those. Anyway, many of old timers remember these long Usenet discussions about T<sub>E</sub>X tricks, where Peter Wilson was prominent with his clever solutions and lucid explanations. Even more people know Peter Wilson for his *memoir* class—a great example of exquisite typesetting. The manual of the class is a great introduction not only to the class itself, but also to the typesetting of books in general.

Starting in 2001, Peter published a column *Glisterings* in *TUGboat*. The articles consisted of fairly

short discussions of tricky L<sup>A</sup>T<sub>E</sub>X problems and their solutions. Some material of this column was based on the discussions in `ctt` and `texhax`. For many years these articles were the favorite part of the journal for me—and, I guess, for numerous other readers.

Now TUG has published these columns in a book, and one can reread them collected together. After doing this myself, I recognized how much my T<sub>E</sub>X style was shaped over the years by them. It was a joy to recall the times when I discovered these little gems of T<sub>E</sub>X programming—and to find the things overlooked at the first reading.

The subjects in the articles vary from complex problems like string parsing to the textbook-like explanation of subjects like the ways of defining new macros in T<sub>E</sub>X and L<sup>A</sup>T<sub>E</sub>X. The columns about paragraphs and their shapes are probably among the most useful in the book. The T<sub>E</sub>X way of dealing with paragraphing is rather complex, and Peter explains it with his characteristic lucidity and clearness. Besides T<sub>E</sub>X and L<sup>A</sup>T<sub>E</sub>X, the book discusses fonts, ornaments and other printer devices, MetaPost (the image on the cover, a *spidron*, is created with MetaPost; see Figure 1), and many other topics. As another proud owner of Lanston Type Company's *LTC Fleurons Granjon* (LTC is now a division of P22 Type Foundry, [p22.com](http://p22.com)), I was especially interested in the chapter about using the Fleurons in L<sup>A</sup>T<sub>E</sub>X. It gives very useful advice on getting the glyphs aligned.

Of course T<sub>E</sub>X has changed over these years. While some problems discussed in the book now have somewhat more straightforward solutions (for example, string manipulations today would be probably based on the `xstring` package), it is surprising how much of the book is still very relevant today and is required reading for an aspiring T<sub>E</sub>Xnician.

The style of Peter's writing is fascinating. The author's subtle British humor makes the reading pleasant and far from the dry stuff filling many technical books. The reader meets this self-deprecating humor on the first page of the Introduction:

For many years Jeremy Gibbons edited a very successful column in the T<sub>E</sub>X Users Group journals *T<sub>E</sub>X and TUG NEWS* and *TUGboat* called *Hey—It works!* [52]. I learnt much from this but apparently not enough to decline when asked to take over the column. On the other hand I have learnt to my cost that the quickest way to get a correct answer to a question on the `comp.text.tex` (`ctt`) newsgroup is to give an incorrect answer. In order not to sully Jeremy's reputation my first thought was to change the title to *Hey—It might work* but after some consideration the new title is

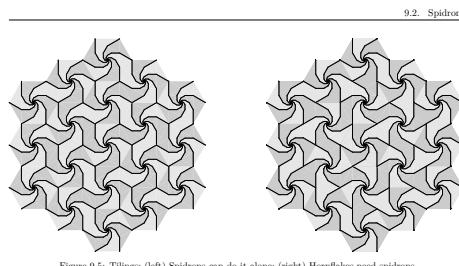


Figure 9.5: Tilings: (left) Spidrons can do it alone; (right) Hornflakes need spidrons

What was missing from this article was any hint as to what those ‘right combinations’ of folds might be to create these effects. After some searching on the web I found the following remarks by Erdely [40].

I folded every second edge, reaching to the centre of the created hexagon in the given Spidron system, as a spine and folded every first edge as a groove. The resulting relief surface, under the impact of an external deflection, follows the rules of how simple linear displacements, such as those produced with an accordian; instead, the edges between the vertices and the centres of the original hexagonal system move in a vortex within each hexagon.

After a lot of cogitation and physical experimentation I came to believe that among the ‘right combinations’ are the ones shown in Figure 9.6, which shows half a hexagon with three semi-spidrons. The dotted lines indicate ‘valley’ folds (paper on either side of the fold is raised), while the full lines indicate ‘mountain’ folds (paper on either side of the crease is bent down).

If you want to create a large construct for folding, here is the code for generating the spidron tiling shown in Figure 9.5. You can, of course, modify this to meet your needs!

```
% glstsp.map MP spidron figures
% earlier pictures
beginfig(5); spidron tiling
```

31

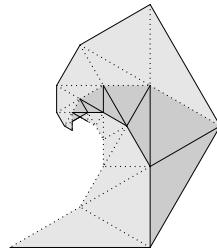


Figure 9.6: Folding. [Editor’s note: We gratefully acknowledge Jeremy Gibbons’ paper ‘Dotted and dashed lines in METAFONT’, TUGboat 16(3) (1995), <https://tug.org/TUGboat/tb16-3/tb46gib.pdf>, which aided us in finalizing this figure, and a fascinating read in itself!]

Figure 1: A page from the column about spidrons

as you see it earlier — *Glisternings* — implying that there might be some dross among the nuggets.

and continues to walk along with it during the journey through the book.

Another part of the book’s charm are the epigraphs throughout. They are funny and wise, providing a surprising counterpoint to the point of the text. For example, the section about changing the layout starts with a note by Samuel Johnson,

*Change is not made without inconvenience, even from worse to better.*

The section about the “superstitious” version of enumerate designed to eliminate an item no. 13 is accompanied by the apt passage from George Orwell’s 1984:

*It was a bright cold day in April, and the clocks were striking thirteen.*

The book is lovingly typeset by Peter, reminding us that the author is a TeX wizard and typesetter of a very high caliber. The techniques discussed in the book are illustrated by the book itself; for example, a page of the column about ornaments has a nice frame around it (Figure 2).

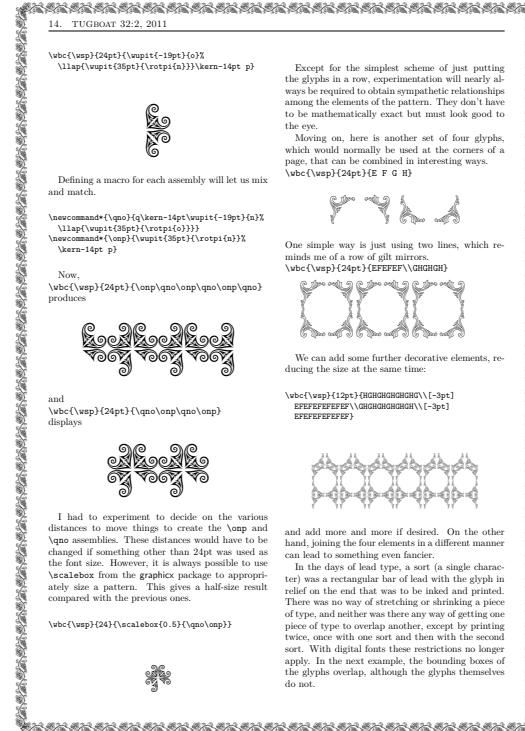


Figure 2: A page from the column about ornaments

When compiling the columns in the book, the author gathered the references from each into a joint bibliography, and added an index — a very useful device for a compendium of disjoint materials like this one. He also rearranged epigraphs and deleted the repeated ones. The resulting book is a welcome addition to a library of a (La)TeX student or even TeXnician — indeed, even for those of us who have a collection of *TUGboat* issues for the last twenty years. I wholeheartedly recommend it.

*Glisternings* is the third book published by TUG, after a volume of interviews (2009) and the 2<sup>5</sup> anniversary collection of papers (2010). This is a fitting continuation of the series. They are all available from [tug.org/books](https://tug.org/books), as well as in the general online stores. I hope that we keep publishing the books under our own imprint — perhaps next time without a 10 year hiatus.

◇ Boris Veytsman  
Systems Biology School,  
George Mason University  
Fairfax, VA  
`borisv (at) lk dot net`  
<http://borisv.lk.net>  
ORCID 0000-0003-4674-8113

## Historical review of **T<sub>E</sub>X3**

Peter Flynn

### Abstract

While clearing out cupboards during lockdown, I came across several oddities, including this review of **T<sub>E</sub>X3** which .exe magazine asked me for in April 1991, during negotiations into being taken over. They subsequently collapsed as a print publication, and the article was therefore never published, meaning it's still my copyright. It was interesting to see what I had picked up on for them, as they had asked me to write it from the point of view of a new **T<sub>E</sub>X** user with experience in other systems.

It referenced five figures which were long since lost, but which I have reconstructed from memory and the descriptions. The original was formatted in unmarked monospace, according to the publisher's requirements. This version has been used without substantive textual change for *TUGboat*, but the plaintext symbolic notations like `\TeX\` have been replaced with the genuine logos, acronyms and URIs have been marked, some previously unnoticed typos have been corrected, and oddities of plaintext markup have been converted to **L<sub>A</sub>T<sub>E</sub>X**.

Most companies, addresses, electronic access, etc., mentioned no longer exist. The references are kept as a matter of nostalgia and history. The article begins below the rule.

*The grand-daddy of DTP systems offers a radically different approach to typography on the desktop.*

*Peter Flynn investigated and found it still has a lot to offer.*

### Introduction

Out of the current soup of DTP systems, a few leaders have emerged among the visual packages. *Ventura*, *Pagemaker*, and *Quark XPress* each has its own benefits and peculiarities, as any user will have discovered, but there is another form of DTP altogether, known as 'logical', which is where systems like **T<sub>E</sub>X** fit in.

Contrary to popular opinion, DTP was not invented in the mid-80s on the Apple Mac, but in 1978 on a DEC-20 minicomputer. The program was **T<sub>E</sub>X**, and it allowed users for the first time to produce printers' quality typesetting from their terminals, using the new 'laser-beam' printers.

Heady stuff indeed at the time, given that word-processing had barely yet been given a name.

The difference between visual and logical systems is straightforward enough once you understand

what it is you're trying to do (the terms 'visual' and 'logical' were coined ca. 1988 [4]).

Visual systems rely on the operator's skill in manipulating a bitmapped screen image, usually by hand, eye and mouse. The relationship between text structure and appearance is undefined, or specified by simple tags in style sheets. The method allows freehand adjustment to appearance, such as positioning, scaling, distorting or overlaying of graphics and text, but places a high degree of reliance on manual dexterity and the visual judgment of the user. Each page is normally made up individually, so that any knock-on effect of changes must also be attended to.

Logical systems rely on the author's or editor's skill to bind structure tightly to appearance, using commands specified in a style file or embedded in the text to define positioning and typography. The text file is edited externally, and processed in its entirety to ensure the effect of changes is properly accounted for from page to page. Adjustments to appearance are made by changing the commands in the text, and by the use of fully programmable macros. This largely removes the need for manual or visual intervention in long or repetitive texts, or in applications where automation, regularity or dimensional accuracy is required.

### Origins

**T<sub>E</sub>X** is, in effect, a document compiler, taking a piece of source 'code' (your text) and acting on the instructions you embedded in it. The original program was written by Don Knuth to typeset a new edition of his famous *Art of Computer Programming* series, because ordinary commercial typesetters either couldn't handle the technical matter, or charged too much. (The lowered 'E' in the **T<sub>E</sub>X** logotype distinguishes it in print from **TEX**, an old Honeywell editor, for copyright reasons, and emphasizes the relationship with typography.)

He generously made **T<sub>E</sub>X** publicly available, so it can be had in commercial and non-commercial versions for almost every machine in existence from Apple IIGS, Amiga, and Archimedes, up through PC and Mac, to Sun and other UNIX systems, VAX/VMS, IBM and other mainframes, and even the Cray (that should shift some lead!). All implementations are compatible with each other, differing only in memory capacity and speed, depending on the operating environment. This level of availability must make it the most widely-spread DTP system in use: certainly in today's increasingly networked environment it is a significant factor to be able to establish completely compatible DTP across almost every platform with no need for investment in special hardware.

$\text{\TeX}$  has had a mixed reception in the past. It has variously been criticised for being difficult to use, lacking graphics, only having one typeface, for not being WYSIWYG and for not being a visual-based system (about as sensible as criticising a fish for not being a chicken). Part of the reason is that, coming from the public domain, there was no one to trumpet its abilities or explain its working. Some reviewers who should know better have not distinguished between visual and logical systems, and some were making their comments based on old versions, hearsay, and incomplete information. However, as we shall see, it is in fact quite easy to use, can handle plenty of graphics, lots of typefaces, and has some of the best screen displays around.

Quality is one area that has always been commented on favourably, and it is carefully guarded: for an implementation to call itself  $\text{\TeX}$  it must pass a stringent test specified by Knuth.

## Availability

Perhaps one of the reasons for the ‘well-kept secret’ ethos surrounding  $\text{\TeX}$  is the fact that the original source code was placed in the public domain. Anyone can have this code for the asking: it was written in WEB, a ‘literate programming language’ also devised by Knuth, which produces Pascal source. There are many commercial versions of  $\text{\TeX}$  as well, written by companies or individuals who have optimised the code for specific operating systems, and ship the system as a package. Because of its popularity and heavy use in research and technical typesetting, and in publishing trade and the academic field, there has been little need for glossy full-colour whole-page adverts for it in magazines aimed at the domestic or office market. This is perhaps a pity, as many visual DTP users who need reliable, accurate and automated formatting are still doing their work manually, unaware that  $\text{\TeX}$  exists!

Those who have anxiety attacks at the mention of the phrase ‘public domain’ can take comfort in the fact that it is the source code itself which is available, so if you have misgivings about viruses, you can check it and compile it yourself. There are no known instances of an infected copy of  $\text{\TeX}$  being distributed, and all the public versions available from the regular sources are checked out before being made available. If in doubt, of course, buy a commercial copy.

My review commercial copy (PC- $\text{\TeX}$ ) came from Uni $\text{\TeX}$  Systems (for details see end) and my public-domain ones ( $\text{em}\text{\TeX}$  and  $\text{SBT}\text{\TeX}$ ) came from the  $\text{\TeX}$  server archive ( $\text{TEX.AC.UK}$ ) at Aston University, Birmingham, via the email network. I also used

an older commercial VAX/VMS version (Kellerman & Smith) and a PD one for the Macintosh called Oz $\text{\TeX}$ . I didn’t have a UNIX machine available, so I was unable to investigate UNIX-flavoured  $\text{\TeX}$  personally, but given the rigorous quality control, it is reasonable to assume it performs as claimed.

There are other public-domain and shareware versions too, from a variety of bulletin boards (e.g. CIX), user groups and file server hosts on the wide-area networks all over Europe and the United States. UK, Irish and continental European users of normal internetwork email can order from Aston or Heidelberg (for details see end) but if you’re trapped on BT Gold or EirMail, forget it and contact your national User Group. In addition there are other regular commercial versions such as *Textures* and *Turbo $\text{\TeX}$*  (for the Mac and PC respectively) and a low-cost commercial version for the PC (DOST $\text{\TeX}$ ).

## Installation

$\text{\TeX}$  for desktop machines needs about 5MB of disk space to live in, and runs in 512KB of memory. On larger machines, where there has traditionally been more disk space, implementations tend to spread themselves around a bit more. If you want to add more fonts, design packages, CAD, specialist macros, foreign-language hyphenation and the endless other goodies, you will need more room. On the other hand, if all you want to do is publish, say, a typeset database listing, you can trim right down to just over 1MB (plus your own data/text storage, of course).

PC- $\text{\TeX}$  came in the usual style of PC software binder: the INSTALL routine worked perfectly and took around 15 minutes to unpack everything from the nine disks.  $\text{em}\text{\TeX}$  arrived as several .B00 archive files (encodings into printable characters which allow 8-bit binary to traverse the 7-bit email networks) which DEBOOed and unZIPped (with the -d option) to recreate all the files and the whole directory structure, in about 20 minutes. I wish some other software I could mention unpacked with even half the care and intelligence that has been spent on organising both these versions. The Mac version I unpacked in a similar manner from BINHEXed STUFFIT archives without problems. The VAX software originally came on standard half-inch magtape and took rather longer, because of the need to establish logical names and pathways to cater for multiuser operation.

Drivers are available for most printers (i.e. those emulating IBM Graphics and ProPrinters, Epson FX and LQ, NEC Pinwriter, PostScript and HP LaserJet) and you can get or make drivers for almost anything which puts marks on paper, even a fax: if you have

something esoteric, there is generic driver code available for you to roll your own. There's even a driver for a CalComp pen plotter if you want letters several feet high! Because *T<sub>E</sub>X* produces an output file which is entirely device-independent, you aren't tied to using any particular supplier's driver, or even any particular machine.

*T<sub>E</sub>X* itself knows nothing about fonts except the height and width of each character, plus a few other parameters, which it reads from font metric files. The print drivers, however, normally use bitmap font files, not outlines, so you need to order the right resolution with your printer driver (usually 180, 240, 300 or 360 dpi depending on your printer). Using bitmaps is a two-edged sword: you tend to get far better quality, but you are restricted to those sizes you have on disk. However, there is a companion type-design program for *T<sub>E</sub>X* called **METAFONT**, which can create additional optimised font files at any resolution for any dot-matrix or laser printer (and some typesetters). **METAFONT** is well worth having if you want extra odd font files at specific sizes, and if you're into type design, this is probably one of the most powerful tools around. If you use PostScript, of course, you don't need any bitmaps, just the font metrics. I shall have more to say about fonts later.

```
\noindent This is a very short test, to make
sure the program is working correctly. This
paragraph starts flush left, and shows the
appearance of {\bf bold face} type.
```

```
This paragraph is indented, and shows the
appearance of {\it italics}. It contains the
math formula $z*n=x*n+y*n$. Such a formula
might also be displayed $$z*n=x*n+y*n$$ to
make it more prominent.
```

```
\bye
```

Figure 1: Source text of the file TEST.TEX

## Documentation

The manual for all implementations of *T<sub>E</sub>X* is *The T<sub>E</sub>Xbook* by Donald Knuth, published by Addison-Wesley [2]. This explains everything, in progressively finer detail. The early chapters are excellent, and can get even a total novice producing the goods very quickly. Learning *T<sub>E</sub>X* is not difficult, but like any DTP system, it is nevertheless not trivial, and the later chapters need some careful reading, as there is a lot in them. There are plenty of worked examples, though, far more than in any other manual I have ever seen (you can never have enough, in my view).

The documentation accompanying PC-*T<sub>E</sub>X* is a more digestible spiral-bound manual which summarises all the introductory matter of *The T<sub>E</sub>Xbook* and looks less forbidding for a beginner [5].

The PD versions came with documentation on how to set up and run the programs (in most cases better explained than most 'professional' manuals) but they do assume you have *The T<sub>E</sub>Xbook* for details of how to format text.

For the user who just wants to produce something with a relatively simple standardised layout, the **L<sub>A</sub>T<sub>E</sub>X** macro package which comes with *T<sub>E</sub>X* is the easiest route, as it has several carefully-designed sample document styles. The **L<sub>A</sub>T<sub>E</sub>X** book which describes these is also an Addison-Wesley publication but comes free with PC-*T<sub>E</sub>X* [3].

There are some very good introductory booklets available free as *T<sub>E</sub>X* input files, so you just process and print them. The best two I found were *A Gentle Introduction to T<sub>E</sub>X* by Michael Doob [1] and *Introduction to T<sub>E</sub>X on VAX/VMS* by Joe St Sauver [6]. This last one sounds a bit operating-system specific, but if you just ignore the VMS bits it is a very good guide. There are others covering the many add-ons to *T<sub>E</sub>X*, again usually supplied on disk as .TEX files for you to print yourself.

This is a very short test, to make sure the program is working correctly. This paragraph starts flush left, and shows the appearance of **bold face** type.

This paragraph is indented, and shows the appearance of *italics*. It contains the math formula  $z * n = x * n + y * n$ . Such a formula might also be displayed

$$z * n = x * n + y * n S$$

to make it more prominent.

Figure 2: Output of TEST.TEX (HP LaserJet III)

## Operation

*T<sub>E</sub>X* operates in a radically different manner to visual DTP. You create your document in a plain ASCII file, embedding formatting commands in the text. When you run the program, it processes your file and makes a compact, portable device-independent typeset file, which is used by the preview and print drivers to produce the output. Because it doesn't use a graphic display during processing, repetitive production jobs such as database publishing can run unattended once they are set up.

A test file (`test.tex`) which demonstrates the method of operation is supplied as standard (see

Figure 1 and Figure 2 for the input and output). The DVI (DeVice-Independent) file can be printed on any supported printer without the need to reprocess your source text: just run the relevant print driver. The concept is similar to PostScript, in that once you have the final version off your laser or dot-matrix printer, you don't have to reformat or reprocess for a higher-resolution device. Unlike PS, however, you are not restricted to specific output devices or fonts.

All the formatting commands are mnemonic, so they are fairly straightforward to learn. They all begin with a backslash (e.g. `\parindent=2em` or `\baselineskip=15pt`, see Figure 1). The use of an ASCII file means you can continue to use your favourite editor (or wordprocessor in 'non-document' or 'ASCII export' mode), so there are no new menus or keystrokes to learn. No editor is supplied, as it is assumed that every machine already has one in some form or another. I'm uncertain about this: although it is easy to get many excellent editor/wordprocessor programs, it surely could have been possible to throw in a good public-domain editor, or license a shareware one, even if only for beginners. I suppose you could even use EDLIN if you were a masochist.

If you've got existing wordprocessor files, there are converter programs from WordPerfect and MS-Word into  $\text{\TeX}$  and  $\text{\LaTeX}$ .

As with any DTP system, once you go beyond elementary formatting you do need to be aware of what you want to do. Typographical training is a much-neglected part of office life so far, although it is noticeably easier to get good typographic quality in your output using  $\text{\TeX}$  than with some visual systems. However, as I mentioned earlier, there are so many predefined layouts available that many users don't bother to delve into the deeper recesses of  $\text{\TeX}$ 's abilities.

The macro facilities mean there is a lot of underlying power available.  $\text{\TeX}$  is in fact a typographical programming language, which means you can program decision-making `\if` statements to modify the appearance depending on the text being processed. The effect of this is to dramatically increase the reusability of your text: using  $\text{\LaTeX}$ 's predefined styles, for example, you can turn a text file into a business report by marking the component parts (title, sections, subsections etc.) and adding three commands at the top and one at the bottom. If your analysis was so good you wanted to turn it into a chapter of your new book, change the document style `report` to `book` and that's it. To publish it as an article, change `book` to `article`, and change the chapters into sections. This is the whole essence of logic-based DTP: once the structure of the text is

marked, formatting and reformatting becomes relatively trivial—just change the macro definitions.

There are some tricks and traps of course. One small confusion for beginners arises over the use of curly braces:  $\text{\TeX}$  uses them to group together text which you want treated in a particular way, so for example, to italicise text you type `{\it your text}` in curly braces, with the '`\it`' command immediately inside the opening brace: this restricts the effect of italics to the text in the braces. So far so good. But curly braces are also used to delimit arguments, so for example, typing `\centerline{some text}` centres the text between the margins. There are very cogent reasons for this, but it means a second or so's thought until you get used to it.

Once you've processed your file, you will want to see the results. The WYSIWYG screen in em $\text{\TeX}$  is one of the best I have seen. It is configurable for all the conventional PC screens from mono CGA to the 64/256 colour VGA display, and on the higher resolution devices it uses grey-scaling to give the image better definition. You can shrink the image small enough to fit two A4 pages on a single screen, or enlarge it big enough to see only a few words at a time. An on-screen ruler lets you measure dimensions, and the image can be rotated, mirrored and inverted. Despite the fact that  $\text{\TeX}$  drivers use bitmap fonts, the previewer does not require its own set of fonts at screen resolution: it can use whatever you have around for your printer, reducing your disk storage needs. Other suppliers also have very good previewers, especially the Preview program from Arbortext.

The print drivers have similar facilities for positioning the output on the page, so it is possible to print portrait or landscape, mirrored or inverted, even on dot-matrix printers. You can easily print selected pages in any order, with multiple pages per sheet in various orientations, so it is possible to make booklets correctly paginated for folding straight off the printer.

The only thing that causes an initial stumbling block is the sequence of edit-process-view-print. This is common to all logical systems, in that the entire file has to be processed before it can be seen. The reason is partly historical, in that  $\text{\TeX}$  grew up before bitmapped screens were commonplace; and partly inherent in a logic-based system, in that type placement on the page naturally depends on what fit onto previous pages. PC- $\text{\TeX}$  does in fact sell a version which displays as you go, and similar versions are available for the Amiga and some other fast machines, but these are very much the exception.

## Timings

**T<sub>E</sub>X** is fast: I clocked just over one-and-a-half seconds per page on a 16MHz 80386 clone with 640KB and a 28ms unfragmented hard disk for processing plain continuous text in em**T<sub>E</sub>X**. By comparison, PC-**T<sub>E</sub>X**'s 80386-specific executable on the same machine but using 2MB of memory gave me under half a second per page on the same file.

Printing speed is limited by the throughput of the printer itself. The HP LaserJet driver printed at the full eight pages a minute, with only a brief pause at the start to download the font glyphs.

Printing full-page graphics on a dot-matrix printer is always tedious, but I counted 34 seconds per A4 page of solid text on an Epson LQ800, which is acceptable for short drafts.

em**T<sub>E</sub>X** runs very happily under DesqView/386, my own preference for a working environment, with PC-Write in one window, the **T<sub>E</sub>X** engine in another and the WYSIWYG screen in another (plus my usual assortment of comms, spreadsheet and database). I haven't tested it under Windows, but there's a Windows (Microsoft Paintbrush-style) screen driver in em**T<sub>E</sub>X**, and the **T<sub>E</sub>X** program and print drivers should cause no problems as DOS tasks.

## When something goes wrong

**T<sub>E</sub>X**'s error messages are explicit but technical. Pressing H when it pauses at an error gives some further explanation, and often points you to the relevant section of *The T<sub>E</sub>Xbook*. It is assumed that you have read some of this, or similar explanatory documentation, because otherwise a message such as 'Overfull \hbox at line 432' is not going to mean much (in fact it refers to a justification or hyphenation problem, the h(horizontal) box being the page element it is trying to justify).

'Errors' in this sense means one of two things: either you have mistyped a command word, which is easy to correct, as it tells you what and where; or there is a fault in the logic of your instructions, which is harder to spot. Unfortunately, computers cannot guess your intentions, although **T<sub>E</sub>X** makes a damn good try. Missing a closing curly-brace is a common typing error, and so is forgetting to turn off a special mode, such as mathematical setting, but this is not logically determinable until your text tries to do something that **T<sub>E</sub>X** knows cannot be done in whatever mode you were in, like finding a paragraph-end in the middle of text supposed to be centred. All the program can do is report what went wrong and where: it's up to you to find out why, and this can be tricky in complex work, as there may be a considerable amount of innocuous text between

the cause of the problem and the actual place where **T<sub>E</sub>X** spotted things going astray.

Justification and hyphenation problems are rare in normal text, because **T<sub>E</sub>X** justifies an entire paragraph at one go, rather than line by line. This results in a wonderfully smooth and professional finish compared with the woefully ragged and uneven spacing found in some systems, but there are occasions when you have to fix an obtuse word with a discretionary hyphen.

The most common mistake I made was forgetting to turn off a typestyle such as boldface, resulting in the remainder of the document being in bold type, but that is easy to spot and easy to fix, as it is obvious from the display or printout where the error starts. Omitting or wrongly delimiting an argument causes unexpected results for the unwary: typing \centerline and then omitting any argument and carrying on with the text makes **T<sub>E</sub>X** centre the first character of the next word and then complain that it can't fit the surrounding text on the line. Perfectly reasonable, but a strong case for RTFM (Read The Flaming Manual).

Fortunately the results of RTFM are well worth it: after the learning curve had flattened a little (an afternoon), doing some automated formatting, even including some mathematics, produced such a professionally-formatted page that the idea of going back to placing everything with the mouse by hand is one I can't contemplate.

**T<sub>E</sub>X** has extremely few bugs, as Don Knuth has been offering hard cash to anyone who can find one, starting at 1 cent and doubling on each occasion. As the current rate is only \$40.96 after 12 years, you are not likely to find that many more.

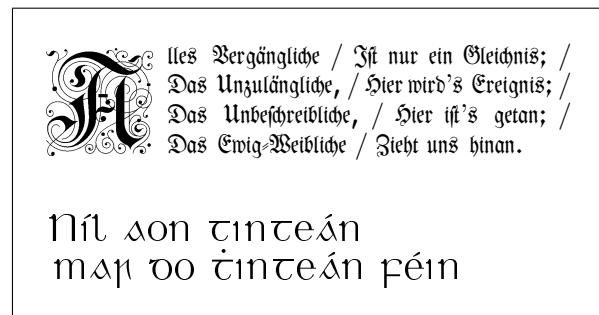
When you need more serious help, therefore, it will be with typography, not bugs, and you will need access to a design or typographic expert fluent in **T<sub>E</sub>X**. In the case of commercial software, this is often provided by the supplier, and either included in the price or charged as support. In the case of public-domain **T<sub>E</sub>X**, you can either call your user group (see details at end), or if you have email access, send a message to one of the many support conferences, where there are hundreds of experts who will gladly help. For specialist or large-scale development, there are also many **T<sub>E</sub>X** consultants who charge normal commercial rates.

## Fonts

The default font is Computer Modern (see Figure 2), a redrawing (by Knuth) of Monotype Modern 8A, and resembling Century Schoolbook, but less bulky. It suits excellently for continuous text, which is what

$\text{\TeX}$  was originally designed for. Because of this, and because Computer Modern has all the scientific and mathematical symbols built-in (many more than in your average DTP system), a lot of  $\text{\TeX}$  users never bother to get away from it. This is a pity, as it has led to the myth that  $\text{\TeX}$  only works with the CM typeface. A browse through *TUGboat*, the  $\text{\TeX}$  Users Group journal, soon explodes the myth:  $\text{\TeX}$  works fine with Bitstream fonts, PostScript, and anything from the Hewlett-Packard downloadable stable, as well as with other METAFONT designs.

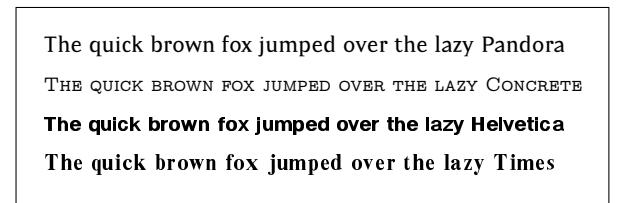
Mind you, you have to pay money for some of these, but I used Bitstream Swiss and Humanist (Helvetica and Optima), the Type 1 Adobe PostScript fonts, and some HP softfonts generated by Glyphix, all without problems (although Glyphix doesn't provide a pound [sterling] sign, being American). These fonts have to be in  $\text{\TeX}$  format, which means a small conversion program from PC- $\text{\TeX}$  Inc. for Bitstream outlines. If you don't have a PostScript printer, you can buy the Adobe fonts as bitmaps from the Kinch Computer Company, and there is a public-domain HPToTeX program available to convert HP downloadables. The Austin Code Works also has a whole stash of fonts already in  $\text{\TeX}$  format, and as I said earlier, if you restrict yourself to a PostScript printer, you can do away with bitmaps entirely, although you lose some of the mathematical and scientific symbols, and the interletter spacing is a little poxy in places.



**Figure 3:** Fraktur with decorated initial, and an Irish text typeface

METAFONT is a programming language for making your own fonts. Font design itself is a decidedly non-trivial activity, but the program can be used with several public-domain font definitions to create font files. There is an increasing number of new fonts being done in METAFONT, and I tested a mixture: some well-established ones like Pandora by Neenie Billawala and Hermann Zapf's Euler (from the American Mathematical Society); Helvetica and Times (more akin to Morison's original, not Times New

Roman) from the MetaFoundry (Dublin, Ohio); and two new ones, a modern Irish typeface by Micheál Ó Searcoid of University College, Dublin; and a new Fraktur and Schwabacher by Yannis Haralambous of CITI Lille, which has some stunning decorated initials (see Figure 3 and Figure 4).



**Figure 4:** Comparison of some text fonts used in  $\text{\TeX}$

Interestingly, *TUGboat* says that the resident 'Wizard of Fonts' is Hermann Zapf himself, so they clearly have some good high-level advice on tap. There are many non-Latin fonts available: Cyrillic, Greek, Devanagari, Arabic, Turkish, and Japanese, even a Tengwar script for Tolkien fans! The ScholarTeX package provides Persian, Ottoman, Pashto, Urdu, Hebrew, Yiddish, Syriac, Armenian, Greek, Ancient Greek and Latin, Fraktur and Schwabacher, Anglo-Saxon, Irish, Glagolitic, Coptic, Calligraphic Arabic and Sanskrit. There's a CM-compatible International Phonetic Alphabet (IPA) from the University of Washington.



**Figure 5:** Difference between design sizes

$\text{\TeX}$  provides the ability to scale any individual font or the whole document to any size (but you still need the bitmaps at the right dot-density in order to print), but *The TeXbook* makes it clear that 5pt type is best done with a real 5pt design, and 50pt type with a 50pt design, rather than by scaling another design size up or down (see Figure 5). This is of course exactly what typographers have been doing since Gutenberg, but some authors of modern systems failed initially to understand the need, and gave only outline fonts. This was partly acknowledged by Adobe's use of 'hints', but it has led to some disastrous results in terms of legibility and

appearance, as anyone who has tried scaling 110pt type down to 5pt can see. Using bitmaps of fonts at different design sizes does place some restrictions on users' disk space, but the selection of Computer Modern shipped with  $\text{\TeX}$  provides a good everyday working subset as a mixture of design sizes and scaled fonts. It is clear that the reason for METAFONT is that you can generate fonts at the size required in a few minutes and then junk the bitmaps when you are finished, if you are short of space (some systems, such as the version for Amiga, do the font generation automatically when you reference a font which is not on disk).

## Graphics

The  $\text{\TeX}$  typesetting engine itself, of course, is not a graphics processor: no logical system is, and even most visual DTP systems make poor drawing packages. Most DTP users are accustomed to scanning artwork and then embedding it into the document, and the same applies to  $\text{\TeX}$ , although you can also use the more elementary line-drawing abilities of the L<sup>A</sup>T<sub>E</sub>X macro package, and the more advanced facilities of PICT<sub>E</sub>X or TEXCAD.

Scanning artwork usually means touching it up in a paintbox program such as PC Paintbrush, and you can then make it into a character in a font file, using a neat little routine from Micro Programs Inc, or if your printer can't handle large downloadable glyphs, you can use the same routine to make a printstream output file which the print driver can handle itself.

This means you can also embed printstream output of some other application in the  $\text{\TeX}$  output (e.g. Encapsulated PostScript, or HP's PCL generated by spreadsheets, business graphics packages, CAD, drawing programs etc.). There are also several routines developed for using half-tones (photographs), in monochrome or colour, although the 300 dpi resolution of a normal laser printer scarcely does them justice.

## Extras

Most implementations of  $\text{\TeX}$  include the macro package L<sup>A</sup>T<sub>E</sub>X, which provides the facilities for structured documents, with variations on autonumbering sections, subsections, paragraphs etc.; automated table-of-contents and indexes; simple diagram and graph-drawing; forward and backward references and a whole stack of other stuff, including BIB<sub>E</sub>X, a bibliographic database program.

There's PICT<sub>E</sub>X, for drawing more complex diagrams such as flowcharts or organisation charts; TEXCAD, a little CAD package which produces L<sup>A</sup>T<sub>E</sub>X

source code as output; and Sli $\text{\TeX}$  for making multi-colour overhead transparency separations. This has recently been extended by David Love of the Daresbury Labs to produce output on colour PostScript printers—what he terms  $\text{\TeX}nicolor$ .

METAFONT, the font-making program, is supplied with em $\text{\TeX}$  plus the complete source code for all the Computer Modern fonts, and some font manipulation tools to go with it. With most commercial implementations, though, METAFONT is a chargeable extra.

The public-domain bolt-ons are legion. In addition to the wide range of styles under L<sup>A</sup>T<sub>E</sub>X, assorted generous people have written and donated macro packages to do circuit diagrams, molecular (hydrocarbon) diagrams, critical text editions, style files for journals, music typesetting, dictionaries, database output, newsletters, calendars;  $\text{\TeX}$  has been used for most things at one time or another over the last dozen years, so there is a wealth of experience to draw on. There are also plenty of agencies and consultants who will design styles in  $\text{\TeX}$  to your specifications.

## Conclusion

The gripes I referred to earlier (lack of fonts, graphics and styles, visual presentation and perceived difficulty of use) seem to have been a problem of users' perception, rather than any deficiency in the  $\text{\TeX}$  system itself.  $\text{\TeX}$  is, however, something of an acquired taste, but has the capability to outperform most other systems in its field in quality, flexibility and automation.

Directly comparing  $\text{\TeX}$  with a visual system is not strictly valid, as the two kinds of DTP are usually aimed at different requirements, although there is a large area of overlap. Those users who feel uneasy with an asynchronous visual feedback are probably more productive with a visual system, but they may have to spend more effort to achieve the same level of quality.

Equally, there are tasks more suited to visual systems than to logical ones: attention-grabbing advertising and newsletters which need to rely on a complex blend of overlapping colour graphics and type for optical appeal are far faster to produce using a visual system than a logical one.

In the end it comes down to suiting the facilities offered to the typographic skill and knowledge of the user, as well as to the demands of the task. Despite the initial appeal of visual systems,  $\text{\TeX}$  is still worth a careful look.

## A Original biography

Peter Flynn is in charge of research and academic computing at University College Cork, Ireland. He has been Technical Consultant to a large City computer bureau, deputy DP manager for one of the UK Training Boards, and a teacher of programming and systems analysis. His hobbies are early music, reading and surfing. He can be reached by email as `pflynn` on BIX and CIX, and through the wide-area networks as `cbts8001@iruccvax.ucc.ie`.

## B Network sources of public-domain software

Email users (e.g. CIX, JANET, HEANET, BITNET, UUCP etc.) can retrieve files from the Aston archive of T<sub>E</sub>Xware: send a one-line electronic mail message to `texserver@tex.ac.uk` saying

`send[tex-archive]00directory.list`

(this is a large file containing a directory of everything in the archive). Individual files can then be retrieved by the same method.

A one-line mail to `listserv@dhdurzl.bitnet` saying just `index` will retrieve the file list from the server in Heidelberg (files are got by sending `get` followed by the filename). Some additional T<sub>E</sub>X material is kept at `mailserv@ymir.claremont.edu`.

All network servers respond to the single-word command `help` by sending you a help file about what is in them and how to access them. BT Gold and EirMail are unfortunately not connected to the international email networks, but Gold 400 is.

In case of difficulty, contact the UKT<sub>E</sub>X Users Group, c/o Aston University Computer Centre, Birmingham (email `uktex@aston.ac.uk`).

## C Commercial software

- PC-T<sub>E</sub>X. UniT<sub>E</sub>X Systems, 12 Dale View Road, Sheffield.
- $\mu$ T<sub>E</sub>X. Arbortext Inc, 535 West William Street, Ann Arbor, Michigan 48103, USA (fax +1 313 996 3573).
- *Textures* (Mac), £350; CM fonts as PostScript outlines, £300. T<sub>E</sub>Xpert Systems, PO Box 1897, London NW6 1DQ (fax +44 71 433-3576).
- For other operating systems, contact the UK T<sub>E</sub>X Users Group (address above) for details of suppliers.
- WordPerfect-to-T<sub>E</sub>X converter, \$249. K-Talk Communications, 30 West First Avenue, Suite 100, Columbus, Ohio 43201, USA (fax +1 614 294 3704).

- TurboT<sub>E</sub>X, \$150; Adobe bitmaps, \$200. Kinch Computer Company, 501 South Meadow Street, Ithaca, NY 14850, USA (fax +1 607 273 0484). Kinch also provides a fax driver for T<sub>E</sub>X.
- Bitmap fonts. Austin Code Works, 11100 Leafwood Lane, Austin, Texas 78750-3464, USA (fax +1 512 258 1342, email `info@acw.com`);
- Capture, \$100; T<sub>E</sub>XPIC (graphics), \$79. Micro Programs, Inc, 251 Jackson Avenue, Syosset NY 11791-4117, USA (tel +1 516 921 1351).
- ScholarT<sub>E</sub>X (out shortly): Yannis Haralambous, rue Breughel 101/11, FR-59650 Villeneuve d'Ascq, France (tel +33 20.05.28.80, email `yannis@FRCITL81.bitnet`).

## D Pull quotes

1. All implementations are compatible with each other, differing only in memory capacity and speed, depending on the operating environment.
2. Omitting or wrongly delimiting an argument causes unexpected results for the unwary: a strong case for RTFM (Read The Flaming Manual).
3. Despite the initial appeal of visual systems, T<sub>E</sub>X is still worth a careful look.

## References

- [1] M. Doob. *A Gentle Introduction to T<sub>E</sub>X*. Aston University, UK: UK T<sub>E</sub>X Archive, Jan. 1990. [ctan.org/pkg/gentle](http://ctan.org/pkg/gentle).
- [2] D. Knuth. *The T<sub>E</sub>Xbook*. Addison-Wesley, Boston, MA, 18th edition, May 1990.
- [3] L. Lamport. *L<sub>A</sub>T<sub>E</sub>X: A Document Preparation System*. Addison-Wesley, Boston, MA, Jan. 1986.
- [4] L. Lamport. Document Production: Visual or Logical? *TUGboat* 9(1):8, Jan. 1988. [tug.org/TUGboat/tb09-1/tb20lamport.pdf](http://tug.org/TUGboat/tb09-1/tb20lamport.pdf)
- [5] M. Spivak. *The PC-T<sub>E</sub>X Manual*. Personal T<sub>E</sub>X, Inc., San Francisco, CA, 01 1985.
- [6] J. St Sauver. Introduction to T<sub>E</sub>X on VAX/VMS. Claremont University, CA: YMIR T<sub>E</sub>X Archive, Jan. 1991.

◊ Peter Flynn  
 Textual Therapy Division,  
 Silmaril Consultants  
 Cork, Ireland  
 Phone: +353 86 824 5333  
`peter` (at) `silmaril` dot `ie`  
[blogs.silmaril.ie/peter](http://blogs.silmaril.ie/peter)



## The Treasure Chest

These are the new packages posted to CTAN ([ctan.org](https://ctan.org)) from August–October 2020, along with a few notable updates. Descriptions are based on the announcements and edited for extreme brevity.

Entries are listed alphabetically within CTAN directories. More information about any package can be found at [ctan.org/pkg/pkgname](https://ctan.org/pkg/pkgname). A few entries which the editors subjectively believe to be of especially wide interest or otherwise notable are starred (\*); of course, this is not intended to slight the other contributions.

We hope this column helps people access the vast amount of material available through CTAN and the distributions. See also [ctan.org/topic](https://ctan.org/topic). Comments are welcome, as always.

◊ Karl Berry  
 tugboat (at) tug dot org

---

### dviware

#### dvivue in dviware

Windows viewer for PDF or DVI.

---

### fonts

#### cmathbb in fonts

Computer Modern math blackboard bold, with complete alphabet and numerals.

#### doulossil in fonts

The Doulos SIL font for IPA typesetting.

#### josefin in fonts

Josefin fonts with L<sup>A</sup>T<sub>E</sub>X support, a geometric sans.

#### plimsoll in fonts

Access to the Plimsoll symbol used in chemistry.

#### spectral in fonts

Spectral fonts with L<sup>A</sup>T<sub>E</sub>X support, a serif face.

---

### fonts/utilities

#### ot2woff in fonts/utilities

Convert OpenType or TrueType to WOFF.

#### t1subset in fonts/utilities

C++ library to subset Type 1 fonts.

#### woff2ot in fonts/utilities

Convert a WOFF file to OpenType or TrueType.

---

### macros/latex/contrib

#### centerlastline in macros/latex/contrib

“Spanish” paragraphs with last line centered.

**decision-table in macros/latex/contrib**  
 Decision tables in Decision Model and Notation (DMN) format.

#### docutils in macros/latex/contrib

Support for Docutils reStructuredText sources ([docutils.sourceforge.io](https://docutils.sourceforge.io)).

#### leftindex in macros/latex/contrib

Left indices (sub/superscripts) with improved spacing.

#### nnext in macros/latex/contrib

Extensions for the gb4e linguistics package.

#### oup-authoring-template in macros/latex/contrib

Template for Oxford University Press journals.

#### qyxf-book in macros/latex/contrib

Book template for Qian Yuan Xue Fu club.

#### realtranspose in macros/latex/contrib

90 degree character transposition.  
**runcode in macros/latex/contrib**

Execute any command line tool and typeset its output.

#### semtex in macros/latex/contrib

Deal with stripped SemantEX documents.

#### swfigure in macros/latex/contrib

Five display modes for handling figures too large to fit on a single page.

#### totalcount in macros/latex/contrib

Typeset total values of counters (from *caption*).

#### xmuthesis in macros/latex/contrib

Thesis for Xiamen University.

---

### macros/latex/required

#### latex-firstaid in macros/latex/required

Late-breaking compatibility fixes for packages, provided by the L<sup>A</sup>T<sub>E</sub>X team.

---

### macros/luatex/latex

#### lua-physical in macros/luatex/latex

Pure Lua library providing functions and objects for computing physical quantities.

#### stricttex in macros/luatex/latex

Strictly balanced brackets; allow numbers in command names.

#### unitipa in macros/luatex/latex

Typesetting TIPA fonts with Unicode input.

---

### macros/unicodetex/latex

Barbara Beeton’s column in this issue (p. 260) describes the new CTAN area **macros/unicodetex**.

#### texnegar in macros/unicodetex/latex

Kashida justification improved wrt *xepersian*.

---

### support

#### light-latex-make in support

A build tool for L<sup>A</sup>T<sub>E</sub>X documents.

## 2021 TeX Users Group election

TUG Elections Committee

The terms of TUG President and ten other members of the Board of Directors will expire as of the 2021 Annual Meeting, expected to be held in July or August 2021.

The terms of these directors will expire in 2021: Karl Berry, Johannes Braams, Kaja Christiansen, Taco Hoekwater, Klaus Höppner, Frank Mittelbach, Ross Moore, Arthur Rosendahl, Will Robertson, Herbert Voß.

Continuing directors, with terms ending in 2023: Barbara Beeton, Jim Hefferon, Norbert Preining.

The election to choose the new President and Board members will be held in early Spring of 2021. Nominations for these openings are now invited. A nomination form is on this page; forms may also be obtained from the TUG office or via [tug.org/election](http://tug.org/election).

The Bylaws provide that "Any member may be nominated for election to the office of TUG President/ to the Board by submitting a nomination petition in accordance with the TUG Election Procedures. Election ... shall be by ... ballot of the entire membership, carried out in accordance with those same Procedures."

The name of any member may be placed in nomination for election to one of the open offices by submission of a petition, signed by two other members in good standing, to the TUG office; the petition and all signatures must be received by the deadline stated below. A candidate's membership dues for 2021 must be paid before the nomination deadline. The term of President is two years, and the term of TUG Board member is four years.

An informal list of guidelines for TUG board members is available at [tug.org/election/guidelines.html](http://tug.org/election/guidelines.html). It describes the basic functioning of the TUG board, including roles for the various offices and ethical considerations. The expectation is that all board members will abide by the spirit of these guidelines.

Requirements for submitting a nomination are listed at the top of the form. The deadline for receipt of completed nomination forms and ballot information is

**07:00 a.m. PST, 1 March 2021**

at the TUG office in Portland, Oregon, USA. No exceptions will be made. Forms may be submitted by fax, or scanned and submitted by email to [office@tug.org](mailto:office@tug.org); receipt will be confirmed by email. In case of any questions about a candidacy, the full TUG Board will be consulted.

Information for obtaining ballot forms from the TUG website will be distributed by email to all members within 21 days after the close of nominations. It will be possible to vote electronically. Members preferring to receive a paper ballot may make arrangements by notifying the TUG office; see address on the form. Marked ballots must be received by the date noted on the ballots.

Ballots will be counted by a disinterested party not affiliated with the TUG organization. The results of the election should be available by mid-April, and will be announced in a future issue of *TUGboat* and through various TeX-related electronic media.

## 2021 TUG Election—Nomination Form

### Eligibility requirements:

- TUG members whose dues for 2021 have been paid.
- Signatures of two (2) members in good standing at the time they sign the nomination form.
- Supplementary material to be included with the ballot: passport-size photograph, a short biography, and a statement of intent. The biography and statement together may not exceed 400 words.
- Names that cannot be identified from the TUG membership records will not be accepted as valid.

The undersigned TUG members propose the nomination of:

**Name of Nominee:** \_\_\_\_\_

Signature: \_\_\_\_\_

Date: \_\_\_\_\_

for the position of (check one):

**TUG President**

**Member of the TUG Board of Directors**

for a term beginning with the 2021 Annual Meeting.

1. \_\_\_\_\_ (please print)

\_\_\_\_\_ (signature) \_\_\_\_\_ (date)

2. \_\_\_\_\_ (please print)

\_\_\_\_\_ (signature) \_\_\_\_\_ (date)

Return this nomination form to the TUG office via postal mail, fax, or scanned and sent by email. Nomination forms and all required supplementary material (photograph, biography and personal statement for inclusion on the ballot, dues payment) must be received at the TUG office in Portland, Oregon, USA, no later than

**07:00 a.m. PST, 1 March 2021.**

It is the responsibility of the candidate to ensure that this deadline is met. Under no circumstances will late or incomplete applications be accepted.

Supplementary material may be sent separately from the form, and supporting signatures need not all appear on the same physical form.

- 2021 membership dues paid
- nomination form
- photograph
- biography/personal statement

**TeX Users Group  
Nominations for 2021 Election**  
P. O. Box 2311  
Portland, OR 97208-2311  
U.S.A.

(email: [office@tug.org](mailto:office@tug.org); fax: +1 815 301-3568)

## TeX Consultants

The information here comes from the consultants themselves. We do not include information we know to be false, but we cannot check out any of the information; we are transmitting it to you as it was given to us and do not promise it is correct. Also, this is not an official endorsement of the people listed here. We provide this list to enable you to contact service providers and decide for yourself whether to hire one.

TUG also provides an online list of consultants at [tug.org/consultants.html](http://tug.org/consultants.html). If you'd like to be listed, please see there.

### **Dangerous Curve**

+1 213-617-8483

Email: [typesetting \(at\) dangerouscurve.org](mailto:typesetting(at)dangerouscurve.org)  
 We are your macro specialists for TeX or L<sup>A</sup>T<sub>E</sub>X fine typography specs beyond those of the average L<sup>A</sup>T<sub>E</sub>X macro package. We take special care to typeset mathematics well.

Not that picky? We also handle most of your typical TeX and L<sup>A</sup>T<sub>E</sub>X typesetting needs.

We have been typesetting in the commercial and academic worlds since 1979.

Our team includes Masters-level computer scientists, journeyman typographers, graphic designers, letterform/font designers, artists, and a co-author of a TeX book.

### **Dominici, Massimiliano**

Email: [info \(at\) typotexnica.it](mailto:info(at)typotexnica.it)

Web: <http://www.typotexnica.it>

Our skills: layout of books, journals, articles; creation of L<sup>A</sup>T<sub>E</sub>X classes and packages; graphic design; conversion between different formats of documents.

We offer our services (related to publishing in Mathematics, Physics and Humanities) for documents in Italian, English, or French. Let us know the work plan and details; we will find a customized solution. Please check our website and/or send us email for further details.

### **Latchman, David**

2005 Eye St. Suite #6

Bakersfield, CA 93301

+1 518-951-8786

Email: [david.latchman \(at\)  
texnical-designs.com](mailto:david.latchman(at)texnical-designs.com)

Web: <http://www.texnical-designs.com>

L<sup>A</sup>T<sub>E</sub>X consultant specializing in the typesetting of books, manuscripts, articles, Word document conversions as well as creating the customized L<sup>A</sup>T<sub>E</sub>X packages and classes to meet your needs. Contact us to discuss your project or visit the website for further details.

### **Monsurate, Rajiv**

India

Email: [tex \(at\) rajivmonsurate.com](mailto:tex(at)rajivmonsurate.com)

Web: <https://www.rajivmonsurate.com>

I have over two decades of experience with L<sup>A</sup>T<sub>E</sub>X in STM publishing working with full-service suppliers to the major academic publishers. I've built automated typesetting and conversion systems with L<sup>A</sup>T<sub>E</sub>X and rendered TeX support for a major publisher.

I offer design, typesetting and conversion services for self-publishing authors. I can help with L<sup>A</sup>T<sub>E</sub>X class/package development, conversion tools and training for publishers and typesetters for book and journal production. I can also help with full-stack web development.

### **Sofka, Michael**

8 Providence St.

Albany, NY 12203

+1 518 331-3457

Email: [michael.sofka \(at\) gmail.com](mailto:michael.sofka(at)gmail.com)

Personalized, professional TeX and L<sup>A</sup>T<sub>E</sub>X consulting and programming services.

I offer 30 years of experience in programming, macro writing, and typesetting books, articles, newsletters, and theses in TeX and L<sup>A</sup>T<sub>E</sub>X: Automated document conversion; Programming in Perl, Python, C, C++, R and other languages; Writing and customizing macro packages in TeX or L<sup>A</sup>T<sub>E</sub>X, knitr.

If you have a specialized TeX or L<sup>A</sup>T<sub>E</sub>X need, or if you are looking for the solution to your typographic problems, contact me. I will be happy to discuss your project.

**Veytsman, Boris**

132 Warbler Ln.  
Brisbane, CA 94005  
+1 703 915-2406  
Email: borisv (at) lk.net  
Web: <http://www.borisv.lk.net>

TeX and L<sup>A</sup>T<sub>E</sub>X consulting, training, typesetting and seminars. Integration with databases, automated document preparation, custom L<sup>A</sup>T<sub>E</sub>X packages, conversions (Word, OpenOffice etc.) and much more.

I have about two decades of experience in TeX and three decades of experience in teaching & training. I have authored more than forty packages on CTAN as well as Perl packages on CPAN and R packages on CRAN, published papers in TeX-related journals, and conducted several workshops on TeX and related subjects. Among my customers have been Google, US Treasury, FAO UN, Israel Journal of Mathematics, Annals of Mathematics, Res Philosophica, Philosophers' Imprint, No Starch Press, US Army Corps of Engineers, ACM, and many others.

We recently expanded our staff and operations to provide copy-editing, cleaning and troubleshooting of TeX manuscripts as well as typesetting of books, papers & journals, including multilingual copy with non-Latin scripts, and more.

**Warde, Jake**

Forest Knolls, CA 94933  
+1 650-468-1393  
Email: jwarde (at) wardepub.com  
Web: <http://myprojectnotebook.com>

I have been in academic publishing for 30+ years. I was a Linguistics major at Stanford in the mid-1970s, then started a publishing career. I knew about TeX from Computer Science editors at Addison-Wesley who were using it to publish products. Beautiful, I loved the look. Not until I had immersed myself in the production side of academic publishing did I understand the contribution TeX brings to the reader experience.

Long story short, I started using TeX for exploratory projects (see the website referenced) and want to contribute to the community. Having spent a career evaluating manuscripts from many perspectives, I am here to help anyone who seeks feedback on their package documentation. It's a start while I expand my TeX skills.

## TUG 2020: Reprise

The screenshot shows the TeX Users Group YouTube channel interface. At the top, there is a logo for 'TeX USERS GROUP' and a title 'TeX Users Group' with '250 subscribers'. Below the title are navigation links for 'HOME', 'VIDEOS', 'PLAYLISTS' (which is underlined to indicate the current view), 'CHANNELS', 'ABOUT', and a search icon. A dropdown menu for 'TUG 2020 Playlists' is open. Below the menu, four video thumbnails are displayed, each representing a different session from the conference:

- TUG 2020 – Lectures**: Steven Matteson, 'The road to Note' (36 views)
- TUG 2020 – Keynote Addresses**: L<sup>A</sup>T<sub>E</sub>X Mathematical Typesetting (2 views)
- TUG 2020 – Interviews**: Interview with a man (2 views)
- TUG 2020 – Beginning L<sup>A</sup>T<sub>E</sub>X Workshop**: A presentation slide (15 views)

Underneath each thumbnail, there is a link to 'VIEW FULL PLAYLIST'.

**Videos for all talks:** [tug.org/l/tug20-video](https://tug.org/l/tug20-video)

YouTube channel: [youtube.com/c/TeXUsersGroup](https://youtube.com/c/TeXUsersGroup)

**Conference schedule and abstracts:** [tug.org/tug2020/program.html](https://tug.org/tug2020/program.html)

**Proceedings (open access):** [tug.org/TUGboat/tb41-2](https://tug.org/TUGboat/tb41-2)

## Calendar

**2020**

- Oct 27–31 Association Typographique Internationale, ATypI All Over, [www.atypi.org](http://www.atypi.org)
- Oct 31 GuIT Meeting 2020, 17<sup>th</sup> Annual Conference, online, hosted by SISSA medialab, Trieste, Italy. [www.guitex.org/home/en/meeting](http://www.guitex.org/home/en/meeting)
- Nov 5–8 AwayzGoose, an online gathering for lovers of type and letterpress, Hamilton Wood Type & Printing Museum and American Printing History Association, Two Rivers, Wisconsin. [woodtype.org/pages/wayzgoose](http://woodtype.org/pages/wayzgoose)

**2021**

- Mar 1 **TUG election:** nominations due, 07:00 a.m. PST. [tug.org/election](http://tug.org/election)
- Mar 10–12 DANTE 2021 Frühjahrstagung and 64<sup>th</sup> meeting, 32 Jahre DANTE e.V., Otto-von-Guericke Universität, Magdeburg, Germany. [www.dante.de/veranstaltungen](http://www.dante.de/veranstaltungen)
- Mar 31 *TUGboat 42:1*, submission deadline.
- May 2–5 CODEX VIII, “EXTRACTION: Art on the Edge of the Abyss”, Richmond, California. [www.codexfoundation.org](http://www.codexfoundation.org)
- Jun ?– Jul ? TypeParis21, intensive type design program, Paris, France. [typeparis.com](http://typeparis.com)
- Jun 30– Jul 2 Nineteenth International Conference on New Directions in the Humanities, “Critical Thinking, Soft Skills, and Technology”, Universidad Complutense Madrid, Spain. [thehumanities.com/2021-conference](http://thehumanities.com/2021-conference)

Jul ??

International Society for the History and Theory of Intellectual Property (ISHTIP), 12<sup>th</sup> Annual Workshop, “Landmarks of Intellectual Property”. Bournemouth University, UK. [www.ishtip.org/?p=1027](http://www.ishtip.org/?p=1027)

Jul 20–21

Centre for Printing History & Culture, CPHC/Print Networks Conference, “A visitor attraction: printing for tourists”, Appleby-in-Westmorland, Cumbria, UK. [www.chpc.org.uk/events](http://www.chpc.org.uk/events)

Jul 26–30

Digital Humanities 2021, Alliance of Digital Humanities Organizations, Tokyo, Japan. [adho.org/conference](http://adho.org/conference)

Jul 26–20

SHARP 2021, “Moving texts: from discovery to delivery”. Society for the History of Authorship, Reading & Publishing. Hosted virtually by the University of Muenster. [www.sharpweb.org/main/news](http://www.sharpweb.org/main/news)

Aug 1–5

SIGGRAPH 2021, Los Angeles, California. [s2021.siggraph.org](http://s2021.siggraph.org)

Aug 2–6

Balisage: The Markup Conference, Rockville, Maryland. [www.balisage.net](http://www.balisage.net)

Aug 18–22

TypeCon 2021, Philadelphia, Pennsylvania. [typecon.com](http://typecon.com)

Sep 10

The Updike Prize for Student Type Design, application deadline, 5:00 p.m. EST. Providence Public Library, Providence, Rhode Island. [prov.pub/updikeprize](http://prov.pub/updikeprize)

Oct 1–3

Oak Knoll Fest XXI, “Women in the Book Arts”, New Castle, Delaware. [www.oakknoll.com/fest](http://www.oakknoll.com/fest)

**Owing to the COVID-19 pandemic, schedules may change. Check the websites for details.**

*Status as of 15 October 2020*

For additional information on TUG-sponsored events listed here, contact the TUG office (+1 503 223-9994, fax: +1 815 301-3568, email: [office@tug.org](mailto:office@tug.org)). For events sponsored by other organizations, please use the contact address provided.

User group meeting announcements are posted at [tug.org/meetings.html](http://tug.org/meetings.html). Interested users can subscribe and/or post to the related mailing list, and are encouraged to do so.

Other calendars of typographic interest are linked from [tug.org/calendar.html](http://tug.org/calendar.html).

## Introductory

- 263 *Michael Barr* / How  $\text{\TeX}$  changed my life
  - math writing, typing, and typesetting
- 260 *Barbara Beeton* / Editorial comments
  - typography and *TUGboat* news
- 265 *Peter Flynn* / Typographers' Inn
  - To print or not to print; Centering (again); New device driver for old format; What's in a name
- 360 *Peter Flynn* / Historical review of  $\text{\TeX}3$ 
  - a 1991 review of  $\text{\TeX}$ , written for those experienced in other typesetting systems
- 259 *Boris Veytsman* / From the president
  - the paradox of early adoption; moving free software forward

## Intermediate

- 368 *Karl Berry* / The treasure chest
  - new CTAN packages, August–October 2020
- 269 *Lorrie Frear* / Eye charts in focus: The magic of optotypes
  - history and design of the optotypes used on eye charts
- 286 *L<sup>A</sup>T<sub>E</sub>X Project Team* / L<sup>A</sup>T<sub>E</sub>X news, issue 32, October 2020
  - *xparse* in the format; hook management system; changes in *graphics*, *tools*, *amsmath*, *babel*
- 281 *Matthew Leingang* / Using DocStrip for multiple document variants
  - step-by-step example of homework assignments with questions, solutions, and more
- 275 *Kamal Mansour* / The Non-Latin scripts & typography
  - examples of the wide range of typesetting and font requirements beyond Latin
- 327 *Luigi Scarso* / Short report on the state of LuaT<sub>E</sub>X, 2020
  - development status and comparison of LuaT<sub>E</sub>X and its relatives: LuaHB<sub>E</sub>X, LuaJIT<sub>E</sub>X and LuaJITHB<sub>E</sub>X
- 324 *Peter Wilson* / Data display, plots and graphs
  - simple example of using tables, scatter plots, line graphs, histograms

## Intermediate Plus

- 341 *Island of T<sub>E</sub>X* / T<sub>E</sub>Xdoc online—a web interface for serving  $\text{\TeX}$  documentation
  - HTTP API for *texdoc* and CTAN topics
- 292 *Frank Mittelbach, Chris Rowley* / L<sup>A</sup>T<sub>E</sub>X Tagged PDF—A blueprint for a large project
  - background and task summary of an extended feasibility study on the L<sup>A</sup>T<sub>E</sub>X web site
- 318 *Vít Novotný* / Making Markdown into a microwave meal
  - freezing Markdown, Minted, and BIBL<sup>A</sup>T<sub>E</sub>X output for single-run processing
- 308 *Nicola Talbot* / *bib2gls*: selection, cross-references and locations
  - selecting, grouping, referencing glossary items with L<sup>A</sup>T<sub>E</sub>X's index facilities
- 348 *Petr Olsák* / OpT<sub>E</sub>X—A new generation of Plain  $\text{\TeX}$ 
  - modern LuaT<sub>E</sub>X format with full Unicode support
- 343 *Michal Vlasák* / MMT<sub>E</sub>X: Creating a minimal and modern T<sub>E</sub>X distribution for GNU/Linux
  - a small T<sub>E</sub>X distribution with LuaT<sub>E</sub>X and OpT<sub>E</sub>X that integrates easily on modern systems

## Advanced

- 299 *Enrico Gregorio* / Functions and *expl3*
  - an introduction to L<sup>A</sup>T<sub>E</sub>X3 programming with functions and variables
- 335 *Hans Hagen* / Keyword scanning
  - peculiarities of parsing keywords, catcodes, and performance
- 337 *Hans Hagen* / Representation of macro parameters
  - considers making unused arguments more efficient and tracing output more consistent
- 320 *Hans Hagen* / User-defined Type 3 fonts in LuaT<sub>E</sub>X
  - constructing a font on the fly that can reference images, other fonts, graphics
- 329 *Hironori Kitagawa* / Distinguishing 8-bit characters and Japanese characters in (u)pT<sub>E</sub>X
  - analysis and solutions for differing interpretations of byte sequences
- 346 *Igor Liferenko* / UTF-8 installations of T<sub>E</sub>X
  - changing T<sub>E</sub>X's encoding to support reading/writing Unicode

## Reports and notices

- 258 Institutional members
- 355 *Charles Bigelow* / Book reviews: *Robert Granjon, letter-cutter*, and *Granjon's Flowers*, by Hendrik D.L. Vervliet
  - detailed review of these two books from Oak Knoll Press on 17th century type designer Robert Granjon
- 358 *Boris Veytsman* / Book review: *Glistering*, by Peter Wilson
  - review of this volume of collected columns from *TUGboat*, published by TUG
- 369 *TUG Elections committee* / TUG 2021 election
- 370 T<sub>E</sub>X consulting and production services
- 372 Calendar