

TUGBOAT

Volume 19, Number 3 / September 1998
1998 Annual Meeting Proceedings

	234	Barbara Beeton / TUG Election Notice
	235	Barbara Beeton / Editorial Comments — A TUG '98 trip report
	237	TUG'98 Attendees
Real World	239	R.W.D. Nickalls / <i>T_EX in the Operating Theatre: An anaesthesia application</i>
Languages and Fonts	242	Janusz M. Nowacki / <i>Antykwa Toruńska: An electronic replica of a Polish traditional type</i>
	244	Richard J. Kinch / <i>Belleek: A call for METAFONT revival</i>
	250	Karel Píška / <i>Georgian scripts</i>
	256	Taco Hoekwater / <i>Generating Type 1 fonts from METAFONT sources</i>
PostScript Topics	267	Bogusław Jackowski, Piotr Pianowski, and Piotr Strzelczyk / <i>Threshing EPS files</i>
	272	Bogusław Jackowski, Piotr Pianowski, and Piotr Strzelczyk / <i>More T_EX-PostScript links</i>
	276	Piotr Bolek / <i>METAPOST and patterns</i>
Tools	284	Hàn Thê Thành / <i>Improving T_EX's Typeset Layout</i>
	289	Daniel Taupin / <i>1tx2rtf: Exporting L^AT_EX documents to Word addicts</i>
	293	Włodek Bzyl / <i>Adding native language support to the CWEB package and the T_EX program</i>
	298	Marcin Woliński / <i>Pretprin — a L^AT_EX₂ϵ package for pretty-printing texts in formal languages</i>
	304	Hans Hagen / <i>The Calculator Demo, Integrating T_EX, METAPOST, JavaScript and PDF</i>
	311	Hans Hagen / <i>Visual Debugging in T_EX, Part 1: The Story</i>
	317	Hans Hagen / <i>Visual Debugging in T_EX, Part 2: The Macros</i>
Futures	318	Karel Skoupý / <i>N^TS: a New Typesetting System</i>
	323	NTG T _E X future working group / <i>T_EX in 2003, Part I: Introduction and views on current work</i>
	330	NTG T _E X future working group / <i>T_EX in 2003, Part II: Proposal for a \special standard</i>
News & Announcements	338	Calendar
	340	TUG'99 Announcement
Production Notes	339	Mimi Burbank / <i>Production notes</i>
TUG Business	341	Institutional members
	342	TUG membership application
Advertisements	343	T _E X consulting and production services
	339	Hug The Lion!
	344	Y&Y Inc.
	c3	Blue Sky Research

TeX Users Group

Memberships and Subscriptions

TUGboat (ISSN 0896-3207) is published quarterly by the TeX Users Group, 1466 NW Naito Parkway, Suite 3141, Portland, OR 97209-2820, U.S.A.

1999 dues for individual members are as follows:

- Ordinary members: \$65; \$10 surcharge if payment received after March 1, 1999, to cover shipment of back issues.
- Students: \$35; \$10 surcharge if payment received after March 1, 1999.

Membership in the TeX Users Group is for the calendar year, and includes all issues of *TUGboat* for the year in which membership begins or is renewed. Individual membership is open only to named individuals, and carries with it such rights and responsibilities as voting in TUG elections. A membership form is provided on page 342.

TUGboat subscriptions are available to organizations and others wishing to receive *TUGboat* in a name other than that of an individual. Subscription rates: \$75 a year, including air mail delivery; \$10 surcharge if payment received after March 1, 1999.

Periodical-class postage paid at Portland, OR, and additional mailing offices. Postmaster: Send address changes to *TUGboat*, TeX Users Group, 1466 NW Naito Parkway, Suite 3141, Portland, OR 97209-2820, U.S.A.

Institutional Membership

Institutional Membership is a means of showing continuing interest in and support for both TeX and the TeX Users Group. For further information, contact the TUG office (office@tug.org).

TUGboat © Copyright 1998, TeX Users Group

Permission is granted to make and distribute verbatim copies of this publication or of individual items from this publication provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this publication or of individual items from this publication under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this publication or of individual items from this publication into another language, under the above conditions for modified versions, except that this permission notice may be included in translations approved by the TeX Users Group instead of in the original English.

Copyright to individual articles is retained by the authors.

Printed in U.S.A.

Board of Directors

Donald Knuth, *Grand Wizard of TeX-arcana*[†]
Mimi Jett, *President*^{*+}
Kristoffer Rose^{*+}, *Vice President*
Don DeLand^{*+}, *Treasurer*
Arthur Ogawa^{*+}, *Secretary*
Barbara Beeton
Karl Berry
Kaja Christiansen
Susan DeMeritt
Judy Johnson⁺
Ross Moore
Patricia Monohon
Cameron Smith, *Volunteer Coordinator*
Petr Sojka
Philip Taylor
Raymond Goucher, *Founding Executive Director*[†]
Hermann Zapf, *Wizard of Fonts*[†]

^{*}member of executive committee

⁺member of business committee

[†]honorary

Addresses

All correspondence,
payments, parcels,
etc.

TeX Users Group
1466 NW Naito Parkway
Suite 3141
Portland, OR 97209-2820
USA

Telephone

+1 503 223-9994

Fax

+1 503 223-3960

Electronic Mail

(Internet)

General correspondence,
membership, subscriptions:
office@tug.org

Submissions to *TUGboat*,
letters to the Editor:
TUGboat@tug.org

Technical support for
TeX users:
support@tug.org

To contact the
Board of Directors:
board@tug.org

World Wide Web

<http://www.tug.org/>

<http://www.tug.org/TUGboat/>

Problems not resolved?

The TUG Board wants to hear from you:
Please email to board@tug.org

TeX is a trademark of the American Mathematical Society.

1998 Annual Meeting Proceedings

TeX Users Group
Nineteenth Annual Meeting
Toruń, Poland, August 17-21, 1998

TUGBOAT

COMMUNICATIONS OF THE T_EX USERS GROUP

TUGBOAT EDITOR BARBARA BEETON

PROCEEDINGS EDITORS MARIUSZ OLKO AND TOMEK PRZECHLEWSKI

VOLUME 19, NUMBER 3

PORTLAND

•
OREGON

• SEPTEMBER 1998

• U.S.A.

1999 T_EX Users Group Election

Barbara Beeton
for the Elections Committee

The terms of the TUG President and of 10 members of the Board of Directors expire as of the 1999 Annual Board Meeting, which will take place in conjunction with the 20th Annual TUG Meeting to be held in August 1999 in Vancouver, Canada.

The current President, Mimi Jett, has stated her intention to stand for re-election. The directors whose terms expire in 1999 are *Kristoffer Rose*, *Don DeLand*, *Barbara Beeton*, Karl Berry, *Kaja Christiansen*, *Susan DeMeritt*, *Judy Johnson*, *Ross Moore*, Cameron Smith and *Philip Taylor*; directors named in *italic* are expected to stand for election to another term. Continuing directors, with terms ending in 2001, are Arthur Ogawa, Patricia Monohon, and Petr Sojka.

The election to choose the new President and Board members will be held in Spring of 1999. Nominations for these openings are now being invited.

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 written mail 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 at least two weeks (14 days) prior to the mailing of ballots. (A candidate’s membership dues for 1999 will be expected to be paid by the nomination deadline.) A nomination form follows this announcement; forms may also be obtained from the TUG office, and electronically via the TUG Web pages at <http://www.tug.org>.

Along with a nomination form, each candidate is asked to supply a passport-size photograph, a short biography, and a statement of intent to be included with the ballot; the biography and statement of intent together may not exceed 400 words. The deadline for receipt at the TUG office of nomination forms and ballot information is **15 March 1999**.

Ballots will be mailed to all members about 30 days after the close of nominations. Marked ballots must be postmarked no more than six (6) weeks following the mailing; the exact dates will be noted on the ballots.

Ballots will be counted by a disinterested party not part of the TUG organization. The results of the election should be available early in June, and will be announced in a future issue of *TUGboat* as well as through various T_EX-related electronic lists.

1999 TUG Election — Nomination Form

Only TUG members whose dues have been paid for 1999 will be eligible to participate in the election. The signatures of two (2) members in good standing at the time they sign the nomination form are required in addition to that of the nominee. **Type or print** names clearly, using the name by which you are known to TUG. 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 1999 Annual Meeting, **August 1999**.

Members supporting this nomination:

1. _____
(please print)

(signature) _____
(date)
2. _____
(please print)

(signature) _____
(date)

Return this nomination form to the TUG office (FAXed forms will be accepted). Nomination forms and all required supplementary material (photograph, biography and personal statement for inclusion on the ballot) must be received in the TUG office no later than **15 March 1999**.¹ It is the responsibility of the candidate to ensure that this deadline is met. Under no circumstances will incomplete applications be accepted.

- nomination form
- photograph
- biography/personal statement

T_EX Users Group FAX: +1 503 223-396
Nominations for 1999 Election
1466 NW Naito Parkway, Suite 3141
Portland, OR 97209-2820
U.S.A.

¹ Supplementary material may be sent separately from the form, and supporting signatures need not all appear on one form.

Editorial Comments — A TUG '98 Trip Report

Barbara Beeton
American Mathematical Society
bnb@ams.org

The 1998 TUG meeting took place at the Nicolas Copernicus University in Toruń, Poland, from August 17–20, hosted by GUST, the Polish Grupa Użytkowników Systemu $\text{T}_{\text{E}}\text{X}$.

Although Toruń itself is a (restored) medieval walled city, the Nicolas Copernicus University is a rather new university outside the walls. Established in the middle of this century, it is named for Toruń's most famous scientific son, the founder of modern astronomy. The campus is largely wooded, with paths and trails leading off in many directions; by the end of the conference, when I had finally figured out the most pleasant routes between the dorms, the dining room and the lecture halls, I was sorry to leave.



The theme of the conference, “Integrating $\text{T}_{\text{E}}\text{X}$ with the Surrounding World”, was quite well explored, from using $\text{T}_{\text{E}}\text{X}$ to generate on-the-spot anesthesia charts for patients’ records, to tools for solving various $\text{T}_{\text{E}}\text{X}$ technical problems, to a wide-ranging discussion of what the future might hold. Although the number of participants was not as great as at some previous meetings, everyone who attended was well supplied with interesting things to think and talk about.

A highlight of the sessions was the presentation by Hans Hagen of his visual debugging tool, developed as an adjunct to $\text{CON}_{\text{T}}\text{E}_{\text{X}}\text{T}$, Hans’ macro system that fills the same niche as $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ — a user interface based on logical markup and a multiplicity of document styles. The audience response to this was overwhelmingly positive; in the voting for the conference “bests” (best content, best presentation,



and overall best paper) there was simply no contest. Hans took home with him several fine old books provided by the local organizers, and the more highly distilled Cathy Booth memorial prize, provided by UKTUG.

One of my personal favorites was the presentation by Dick Nickalls, “ $\text{T}_{\text{E}}\text{X}$ in the Operating Theatre!”; although the text appears in these proceedings, the slides that accompanied it in person added a good bit of immediacy and “local color”.

The last presentation was a joint event led by almost the entire contingent from the Netherlands, a highly interactive session on “futures”: what are the directions of work currently underway, what new features are wanted and needed, is there any hope for standardization of $\backslash\text{specials}$?, ... These topics appear here in the papers entitled “ $\text{T}_{\text{E}}\text{X}$ in 2003”, and are well worth mulling over.

However, not all was business. An evening outing to the castle at Golub-Dobrzyn found us gathered around a bonfire in the courtyard, eating slices of a pig roasted over the fire by two yeomen clad in “chain mail” (actually, loosely knit hooded overshirts, coated with metallic paint). After a tour of the castle, which had been a headquarters of the Teutonic Knights before their ejection from Poland, and later a home to Polish royalty, we were treated to the spectacle of a fire breather (Gilbert van den Dobbelsteen, who has other talents besides his expertise with output device drivers). The evening ended with guitar playing and folksong.



The gracious hospitality of our hosts was surely appreciated. The local committee included Jerzy Ludwichowski (chair), Jolanta Szelatyńska, Stanisław Wawrykiewicz, Marek Czubenko, and Mariusz Czerniak. The programme committee consisted of Bogusław Jackowski, Chris Rowley, Phil Taylor, Jiří Zlatuška and Hans Hagen. Mariusz Olko, Tomasz Przechlewski and Bogusław Lichoński were responsible for producing the preprints. The office staff included the daughters of two committee members as well as several students, all competent, cheerful, and ever-helpful.



Another evening was spent in the newly restored Artus' Hall in the center of Toruń, where we were serenaded by a string quartet, and wined and dined in a grand manner. This hall, a Neo-Renaissance building from the late 1800s, is on the site of an older hall, dating to the 14th century, where a treaty was signed in 1466 ending a 13-year war between the Poles and the Teutonic Knights, and establishing Poland as an independent state.



A number of sponsors contributed to the success of the conference: the State Committee for Scientific Research; several departments of the Nicolas Copernicus University; the International Center for Information Systems and Services, the Faculty of Mathematics and Informatics, and the Faculty of Economic Sciences and Management; Wydział Kultury Urzędu Miasta Torunia; Acer Polska; Artgraf Warszawa; BOP s.c. Gdańsk; VIA Publishing Toruń-Wrocław; Sun Microsystems Poland; the 4allTeX Team; DANTE e.V.; GUTenberg; UKTUG; NTG; GUST.

Thanks to all.

Editor's note: The photos are included for those of you who were unable to attend the meeting. They were taken from the TUG'98 web site <http://www.gust.org.pl/TUG98/photos>, and were provided by Luzia Dietsche. You *must* see them in color!

Mimi Burbank

TeX in the Operating Theatre: An anaesthesia application

R. W. D. Nickalls BSc, PhD, MBBS, FRCA.

Consultant in Anaesthesia & Intensive Care,

Department of Anaesthesia,

City Hospital, Nottingham, UK.

Telephone +44-(0)-115-9691169

FAX +44-(0)-115-9627713

dicknickalls@compuserve.com

Abstract

This article describes the author's experience of using TeX for typesetting the *Anaesthesia Record* as part of an automated data-collection system developed for use in the operating theatre.

Introduction

Since the theme of this year's conference is "Integrating TeX with the surrounding world" I would like to describe my integration of TeX with the world of the operating theatre — specifically with the domain of anaesthesia.

One of the many things that occupies anaesthetists during an operation is documentation. This takes the form of a log of various physiological parameters (see Figure 1), drugs used, blood lost, fluids administered, procedures performed etc., otherwise known as the *Anaesthesia Record*. Since this is generally a hand-written record, the documentation side of things can become rather neglected during busy periods, and consequently, anaesthetists are increasingly using computers to automate the collection of such data. This has many advantages including allowing real-time processing of data, generation of various derived parameters, and greatly enhanced information display facilities.

Collecting and processing the data

Since most monitoring equipment used in Critical Care environments has an RS-232 serial interface the process of data-collection, construction of trend graphics, formatting and typesetting can be automated reasonably easily.

My own system is a menu-driven research application which uses compiled QuickBASIC programs to coordinate the access, display and printing of both real-time physiological data and keyboard inputs. The printing module uses L^AT_EX to typeset the text and graphics to create the *Anaesthesia Record* in a format suited to the hospital notes.

The data from the various anaesthesia monitors is accessed via the serial port using a multiplexing device. Individual parameters are then extracted using the relevant software for each of the various monitors — see [1] for interfacing details relating to particular anaesthesia monitors. Unfortunately there is currently no standardisation with regard to data formats for medical monitoring devices, but this may well soon change with the development of the new international Medical Information Bus (MIB) standard.

During anaesthesia the program accesses and displays all the data in real-time as graphic trends, as well as deriving a number of so-called 'value-added' parameters and processing keyboard entries. At the end of the operation the program typesets the text and graphics to form the *Anaesthesia Record*.

The graphics are created using the excellent *freeware* program GNUPLOT¹ which allows batch processing and will output graphics in L^AT_EX picture format.

Armed with the maximum and minimum values for each of the measured parameters, the program writes the GNUPLOT input files, and then calls GNUPLOT, outputting the graphics in L^AT_EX picture format, and placing them into the appropriate directories. The program then writes the L^AT_EX input .tex file, and then calls L^AT_EX to typeset the text and graphics. Finally the .dvi file is printed and put into the hospital notes. In practice all this is performed locally within the operating theatre, such that the *Anaesthesia Record* is printed and placed in the patient notes just as the patient is returned to the recovery area. Figure 1 shows the graphics page of a typical *Anaesthesia Record*.

¹ http://www.cs.dartmouth.edu/gnuplot_info.html

Advantages of ASCII-based systems

The fact that both T_EX and GNUPLOT use inputs which are ASCII-based has the great advantage that their input files can be written on-the-fly by the coordinating computer program. Such flexibility allows the final text and graphics of the document to be tailored to the data. For example, this allows the axes of graphs to be automatically adjusted depending on maximum and minimum values. Similarly, text layout can be made to vary depending on the particular keyboard entries made during the operation.

Small is beautiful

An automated system for data collection, display and printing has clear advantages over the usual hand-written method; it is certainly a more accurate record, and physiological data can be sampled much more frequently. Furthermore, keyboard entry

of drugs and other information can be made simple and fast by careful design of the interface.

Since this is a specific stand-alone application, it is possible to use a much cut-down version of L^AT_EX consisting only of the essential files, fonts and style options required for the application, with the effect that the size of the printing module can be made extremely small. A not insignificant bonus, therefore, of using T_EX as the typesetting engine is that I am able to make use of old 386 PCs having relatively small hard-drives, which have been discarded by my memory-hungry colleagues!

References

1. Nickalls RWD and Ramasubramanian R (1995). *Interfacing the IBM-PC to medical equipment; the art of serial communication*. Cambridge University Press, Cambridge, UK. pp 402. ISBN: 0 521 46280 0.

ANAESTHETIC SHEET

Theatre 1, City Hospital, Nottingham, UK.

JOHN DOE
 dob 24/01/1925
 Hosp No: 123456789
 Nightingale Ward
 Age: 75

DATE: 18 August 2000
 OPERATION: Laparotomy
 ANAESTHETISTS: RWD Nickalls *et al.*
 SURGEONS: AN Other *et al.*

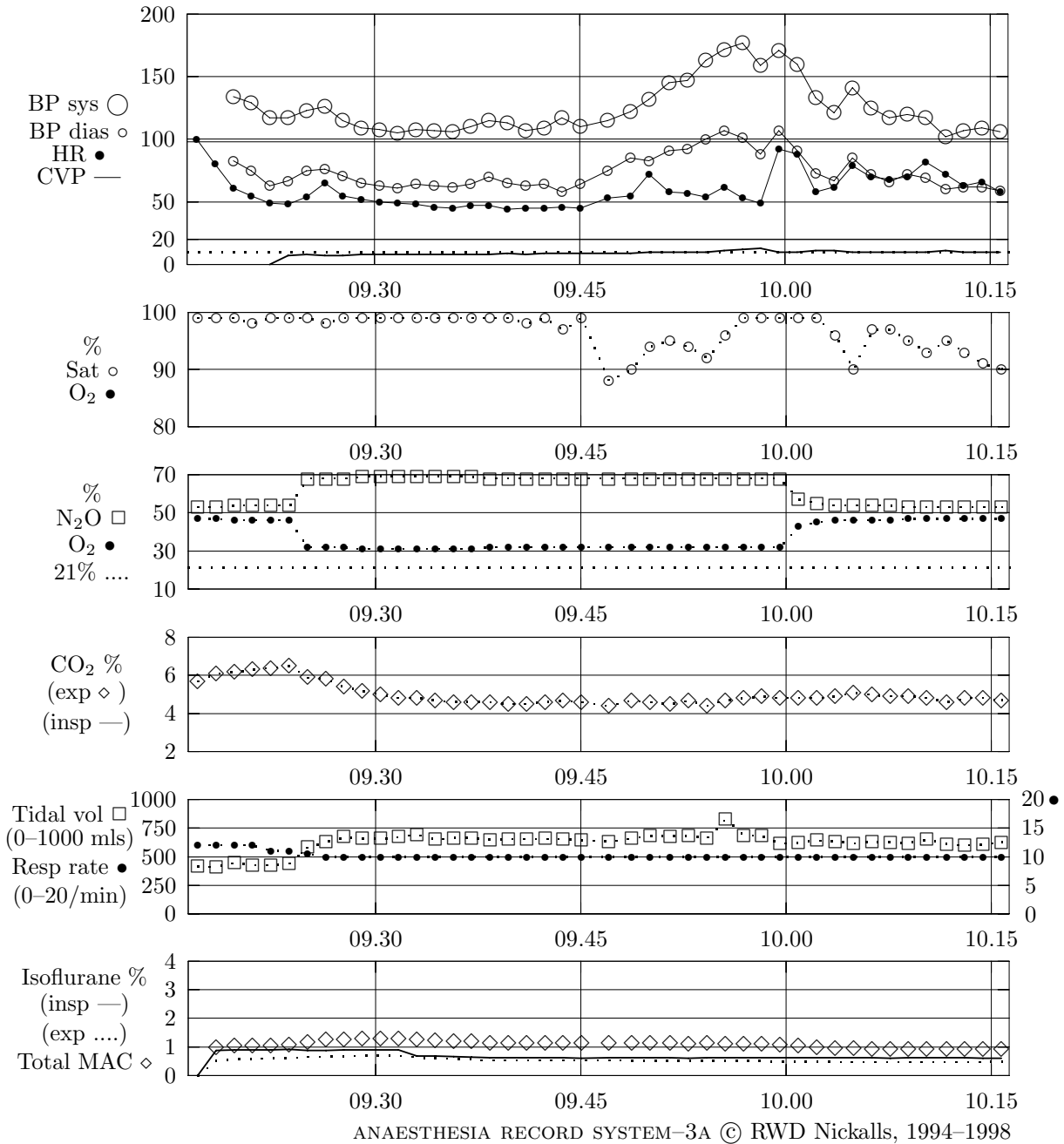


Figure 1: Example of the graphics section of a typical *Anaesthesia Record*. The six graphs are output by GNU PLOT in L^AT_EX picture format. The record shows blood pressure (BP), heart rate (HR), central venous pressure (CVP), oxygen saturation of haemoglobin (Sat), inspired oxygen (O₂), inspired nitrous oxide (N₂O), expired carbon dioxide (CO₂), tidal volume, respiration rate, isoflurane and MAC.

UVWXYZ [] ^ ˇ ˘ a b c d e f g h i j k l m n o p
 q r s t u v w x y z — — “ ” À Á > È Ĩ < Ł Ñ ~ ℓ
 † ‡ Š Š Š ° † Ÿ Ÿ Ź Ź Ź { } § ¨ © ® ÷ € ¡ — ×
 ł ń ± « » ¶ ś ś ś • † Ÿ Ź Ź Ź • “ ” À Á Â Ã Ä Å \ Ç
 È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö × % ù Ú Û Ü Ý | à
 á â ã ä å ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ù ú û ü ý „

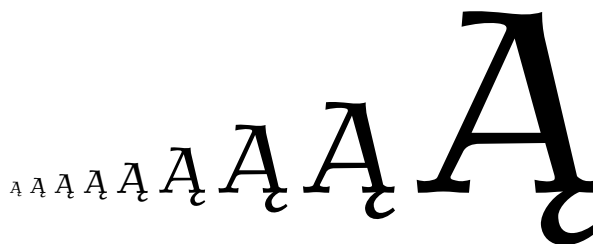


Figure 2: Antykwa Toruńska Italic (anttri) at 6, 8, 10, 12, 16, 24, 36, 48, 96pt

As with the original (metal) version of Antykwa Toruńska, the font contains *three* shapes: regular, italic and bold.

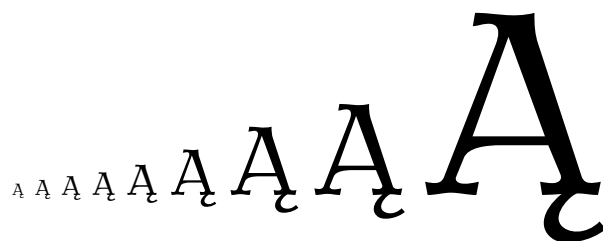


Figure 1: Antykwa Toruńska Regular (anttr) at 6, 8, 10, 12, 16, 24, 36, 48, 96pt



Figure 3: Antykwa Toruńska Bold (anttb) at 6, 8, 10, 12, 16, 24, 36, 48, 96pt

Belleek: A Call for METAFONT Revival

Richard J. Kinch

TRUETEX Software

6994 Pebble Beach Ct

Lake Worth, Florida 33467 USA

kinch@holonet.net

<http://idt.net/~truetex>

Abstract

Despite the importance of mathematical typesetting to the persistent popularity of T_EX, very few T_EX math fonts are available to complement the thousands of available text typefaces. Developing fonts is a very different enterprise from programming other software, requiring different tools and different skills, offering little reward in technical innovation, and often requiring a commercial price to justify the effort. To the corpus of public-domain T_EX software, we contribute *Belleek*, a new set of hand-drawn math fonts to complement Times, published simultaneously in METAFONT, Type 1, and TrueType formats, and compatible with the L^AT_EX `mathtime` package. We describe the difficult process of creating such software with a graphical editor, which motivates a second-look at METAFONT as a practical design tool. We examine Hoenig's METAFONT-based *MathKit* software as a paradigm of fitting math fonts to text typefaces, concluding that METAFONT will become practical only when it gains a visual editor for input and outline fonts for output.

These issues should be vital to mathematical publishing, because meta-math fonts will likely be the only economical source for math fonts to complement most of the universe of text typefaces.

Introducing the Belleek Fonts

Figures 1–3 set forth character-set tables and sample uses of the Belleek math fonts for use with Times text. Figure 4 shows various samples of math-mode usage of the fonts. These fonts are herewith contributed to the public domain, with hopes that T_EX will thereby gain some small measure of flexibility to adapt freely to typefaces other than Computer Modern. The fonts were hand-drawn using Fontographer 4.1, under the following constraints and goals:

1. The character set, encoding, and metrics must match the three math fonts underlying the L^AT_EX `mathtime.sty` package,¹ thus allowing their compatible use with a single `\usepackage{mathtime}` command.
2. All characters must harmonize with the visual style and weight of Times text.
3. The character shapes must be original designs in those cases admitting sufficient latitude for originality, so as to avoid any appearance of infringement of existing proprietary designs.

4. The math symbol shapes (as opposed to Greek letters) should follow the general form of the Computer Modern meta-designs, to the extent possible while still strictly harmonizing with Times. This goal is an experimental excursion into the principle that a meta-typeface might serve as a basis for automatic generation of new math fonts.

Creating Belleek by Hand

Knuth said that success with METAFONT would depend on “collaborative efforts” between “artists and programmers” [5, preface]. One wonders if these classes of people ever meet, because despite the earnest hopes of many, the elegant mathematical and linguistic power of METAFONT has seen little application to font design, and has not been recently used by the commercial type-design industry. Taking in hand once again those beautiful Volumes C and E of *Computers and Typesetting*, and observing the erudition therein, one wonders how such magnificent engines have seen such little use. Have METAFONT and Computer Modern become museum pieces, like some polished-brass steam

¹ Namely the MathTime fonts `mtex`, `mtsy`, and `rmtmi`.

Γ	Δ	Θ	Λ	Ξ	Π	Σ	Υ	Φ	Ψ	Ω	α	β	γ	δ	ϵ
ζ	η	θ	ι	κ	λ	μ	ν	ξ	π	ρ	σ	τ	υ	ϕ	χ
ψ	ω	ε	ϑ	$\var�$	ϱ	ς	φ	\leftarrow	\rightarrow	\dashrightarrow	\dashleftarrow	\leftarrow	\rightarrow	$($	$)$
Γ	Δ	Θ	Λ	Ξ	Π	Σ	Υ	Φ	Ψ	\cdot	$,$	$<$	$/$	$>$	\star
∂	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	\flat	\natural	\sharp	\smile	\frown
ℓ	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	\bullet	\int	\varnothing	\times	Ω

Figure 1: Belleek Math Italic (blmi) 10 pt at 600 dots/inch.

\perp	\cdot	\times	$*$	\dagger	\diamond	\ddagger	\S	\oplus	\ominus	\otimes	\oslash	\odot	\circ	\bullet		
\succ	\equiv	\sqcup	\sqcap	\leq	\geq	\ll	\gg	\sim	\approx	\sqcup	\sqcap	\ll	\gg	\succ	\prec	
\leftarrow	\rightarrow	\uparrow	\downarrow	\leftrightarrow	\nearrow	\searrow	\curvearrowright	\leftleftarrows	\rightrightarrows	\Uparrow	\Downarrow	\Leftrightarrow	\Uparrow	\Downarrow	\curvearrowright	
\int	∞	\in	\ni	Δ	∇	$/$	$_$	∇	Ξ	\lrcorner	\emptyset	\Re	\Im	\top	\perp	
\approx	\sim	\circ	$+$	$=$	\rightarrow	\triangleleft	\triangleright	$=$	$;$	\prime	\prime	\vee	\vee	$-$	\wedge	
\cdot	\sim	\ddots	$_$	NE	NE	NE	NE	NE	NE	NE	NE	\cup	\cap	\oplus	\wedge	\vee
\top	\perp	\lrcorner	\lrcorner	\lrcorner	\lrcorner	$\{$	$\}$	\langle	\rangle	$ $	\parallel	\updownarrow	\updownarrow	\backslash	$?$	
$\sqrt{\quad}$	Π	∇	\int	\sqcup	\sqcap	Ξ	Ξ	NE	NE	NE	NE	\clubsuit	\diamond	\heartsuit	\spadesuit	

NE=Not encoded in font

Figure 2: Belleek Math Symbols (blsy) 10 pt at 600 dots/inch.

$($	$)$	$[$	$]$	$ $	$ $	$[$	$]$	$\{$	$\}$	\langle	\rangle	$ $	\parallel	$/$	\backslash
$($	$)$	$($	$)$	$[$	$]$	$ $	$ $	$[$	$]$	$\{$	$\}$	\langle	\rangle	$/$	\backslash
$($	$)$	$[$	$]$	$ $	$ $	$[$	$]$	$\{$	$\}$	\langle	\rangle	$/$	\backslash	$/$	\backslash
$/$	\backslash	$[$	$]$	$ $	$ $	$ $	$ $	$ $	$ $	$ $	$ $	$ $	$ $	$'$	$'$
\backslash	$/$	$'$	$'$	\langle	\rangle	\sqcup	\sqcap	\S	\S	\odot	\odot	\oplus	\oplus	\otimes	\otimes
Σ	Π	\int	\cup	\cap	\oplus	\wedge	\vee	Σ	Π	\int	\cup	\cap	\oplus	\wedge	\vee
Π	Π	$\hat{\quad}$	$\hat{\quad}$	$\hat{\quad}$	$\tilde{\quad}$	$\tilde{\quad}$	$\tilde{\quad}$	$[$	$]$	$ $	$ $	$[$	$]$	$\{$	$\}$
$\sqrt{\quad}$	$\sqrt{\quad}$	$\sqrt{\quad}$	$\sqrt{\quad}$	$\sqrt{\quad}$	$ $	$[$	\parallel	\uparrow	\downarrow	\prime	\prime	\prime	\prime	\uparrow	\downarrow

Figure 3: Belleek Math Extension (blex) 10 pt at 600 dots/inch.

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)|\varphi(x+iy)|^2 = 0$$

$$\pi(n) = \sum_{m=2}^n \left[\left(\sum_{k=1}^{m-1} \lfloor (m/k) / \lceil m/k \rceil \rfloor \right)^{-1} \right].$$

$$p_1(n) = \lim_{m \rightarrow \infty} \sum_{v=0}^{\infty} (1 - \cos^{2m}(v!^n \pi/n)).$$

Let H be a Hilbert space, C a closed bounded convex subset of H , T a nonexpansive self map of C . Suppose that as $n \rightarrow \infty$, $a_{n,k} \rightarrow 0$ for each k , and $\gamma_n = \sum_{k=0}^{\infty} (a_{n,k+1} - a_{n,k})^+ \rightarrow 0$. Then for each x in C , $A_n x = \sum_{k=0}^{\infty} a_{n,k} T^k x$ converges weakly to a fixed point of T .

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

$$\prod_{j \geq 0} \left(\sum_{k \geq 0} a_{jk} z^k \right) = \sum_{n \geq 0} z^n \left(\sum_{\substack{k_0, k_1, \dots \geq 0 \\ k_0 + k_1 + \dots = n}} a_{0k_0} a_{1k_1} \dots \right).$$

$$\Pi_R \begin{bmatrix} a_1, a_2, \dots, a_M \\ b_1, b_2, \dots, b_N \end{bmatrix} = \prod_{n=0}^R \frac{(1 - q^{a_1+n})(1 - q^{a_2+n}) \dots (1 - q^{a_M+n})}{(1 - q^{b_1+n})(1 - q^{b_2+n}) \dots (1 - q^{b_N+n})}.$$

$$\int_{-\infty}^{\infty} e^{-x \cdot x} dx = \sqrt{\pi}$$

$$X = \sum_i \zeta^i \frac{\partial}{\partial x^i} + \sum_j x^j \frac{\partial}{\partial \dot{x}^j}$$

Figure 4: Math mode samples (after *The T_EXbook*) using Belleek and Times New Roman, 10 pt at 600 dots/inch.

engine, impressive to look at, but long since superseded by higher-powered technology? Or were they perhaps ahead of their time, and not yet harnessed to their potential to create?

Whatever its virtues as a field of human endeavor, working with type as software seems to stifle one's yearning to abstract and perfect a physical enterprise in mathematical form. Instead, it seems to stir the passions for raw, un-parameterized, bare-handed manipulation of perimeters. You want to grab a shape like a piece of hefty rope, not tweeze some bits of code.

Thus fonts today are drawn using direct-manipulation, CAA (computer-aided agony) tools that

somewhat speed the brutish task of digitizing and refining outlines. Big publishers have in-house software, and the small-time designers have GUI software such as *FontLab*, *Fontographer*, and *Type Designer*.

This author is a true believer in languages as a means to use computers. In respect of font design this could hardly be better implemented than through METAFONT. Yet when it came to the practical problem of creating a few fonts in the shortest time, even these near-absolute principles fell to the expedience of the GUI tools. It is indeed faster to just click and drag, just not to be recommended as a steady job, if you value your

sanity. The $\text{T}_{\text{E}}\text{X}$ world would have also been better off with a meta-version instead of a merely-specific version.

As miserable as creating font shapes is, the need to hint fonts for low-resolution use is even more so. Hinting is like undertaking. One applies grisly techniques to preserve the corpse from decay in the presence of trying conditions. Success is achieved when the casual observer comments on how natural the result looks. The Belleek fonts are auto-hinted (let us not carry the grim metaphor any further). Perhaps someone will have the ghoulish expertise to do a proper job on them.

Creating Math Fonts Automatically

Alan Hoenig has taken the right approach in his *MathKit* [2] and *MathInst* [3] packages. In principle we should be able to program (say, in METAFONT) meta-characters for math symbols, and fit them automatically (with, say, METAFONT) to a given text typeface. He exhibits successful applications of this principle, instantiating the Computer Modern character programs with hand-measured characterizations (x-height, stem widths, etc.) of a few typefaces, such as Times, Baskerville, Jenson, and Caslon. Knuth's ancient (in software years) Computer Modern math symbols seem to have been endowed with a sufficient amount of meta-ness to cover a range of target typeface styles. Where meta-qualities are lacking in Computer Modern math, the METAFONT programs can be upgraded.

So if there is great demand for $\text{T}_{\text{E}}\text{X}$ to typeset math in anything-but-Computer-Modern fonts, why has *MathKit* not received popular acceptance? It is not due to any shortcoming in the results, but rather to the utter mess that $\text{T}_{\text{E}}\text{X}$ (and more so $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$) have made of changing fonts, encoding, and styles. What a user wants is not a programming kit containing dozens of components for designing new fonts, but a single command that says "I mean to use Goudy, so please just make it so." Instead, the *MathKit* approach requires an almost superhuman expertise in $\text{T}_{\text{E}}\text{X}$, PERL, NFSS, and not a few other sophisticated tools. This is not to criticize *MathKit* for being overly obscure; in examining its implementation one must admire the economy and efficiency it displays. The source of the complexity is just the nature of the instantiation task.

The simplification of this daunting complexity is not as simple as gathering the output of *MathKit* for a given typeface into a ready-to-run package, creating a little archive that the user can drop in the TDS tree. Because *MathKit* in part depends on METAFONT to rasterize fonts, *MathKit* must neces-

sarily impose several layers of scripts and programs to guarantee that, for example, METAFONT can generate bitmaps for the fonts in the sizes eventually called up in the user's document. METAFONT seems to be the dowdy aunt who is welcomed at first, but then doesn't know when to leave.

Minimizing the user's task requires something more, namely conversion of the *MathKit* instantiation of the meta-math fonts into scalable outlines. This reduces the components of a new style to a few outline font files, a few $\text{T}_{\text{E}}\text{X}$ virtual fonts, and some $\text{T}_{\text{E}}\text{X}$ or $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ macros, all in a ready-to-run distribution. The key, therefore, is the ability to convert METAFONT designs to outlines.

METAFONT: The Flaw

If the ability to convert METAFONT designs to outlines is key, why has it been lacking? Indeed, we reach a startling conclusion: METAFONT is fundamentally flawed, and this flaw has inhibited its acceptance as a font-design tool, namely:

*METAFONT should produce outlines,
not bitmaps, as output!*

Confirming this assertion is the software-tools philosophy. METAFONT is at heart a language for the expression of mathematically abstract shapes. As a properly demarcated tool, METAFONT should convert that abstraction form to another abstraction, and do no more and no less. Knuth's problem with METAFONT in 1982 was that he, his students, and colleagues were unable to practically solve the computational-geometry problem of converting stroked elliptical pens to outlines, and of overlapping shapes to outlines. Instead he relied upon rasterization as an expediency [6].

Also confirming this assertion is the existence and popularity of METAPOST [1], which converts METAFONT code to PostScript code, something closer to (although still not quite) outlines. METAPOST works by intercepting METAFONT's internal data structures before they are bitmaps. In essence it is taking the proper output from METAFONT and expressing it in an intermediate form using the more primitive PostScript language.

METAFONT: The Redemption

If METAFONT should produce outlines, but Knuth sidestepped the problem, then what are we to do? Ask him to try again, but harder? There would seem to be two routes to getting outlines, instead of bitmaps, from METAFONT:

Outlines from Overlapping Shapes. METAPOST converts METAFONT code to PostScript code,

expressing the same overlapping shapes in a more primitive geometric form. MetaFog [4] first exhibited a practical solution to the further task of reducing overlapping, stroked PostScript shapes to non-overlapping outlines. We are tantalizingly close to a “MetaFog 2” that completes the theoretical solution and implements it in robust form. This would allow fast and complete conversion of METAFONT code to non-overlapping outlines, such as are required for outline font formats.

Proper Outlines from Curve-Fitting Polygons or Bitmaps. The MetaFog research attempts to solve a generalization of an already-solved problem, that of removing overlaps in polygons or bitmaps. If we convert the overlapping METAFONT shapes to polygons or bitmaps, then we can apply well-understood algorithms to compute the equivalent non-overlapping polygons or bitmaps. Indeed, in the bitmap domain we have merely described what METAFONT now does, namely, it computes a single bitmap resulting from the rasterization of any number of overlapping shapes.

If we consider a polygon (or bitmap) as a digital sampling of an underlying analog shape, then it would appear consistent with sampling theory that the band-limited analog shape underlying the polygon (or bitmap) should be recoverable, given that we have sufficient resolution and absence of noise in the polygon coordinates (or bitmap pixels). This “given” is assured in the case of METAFONT, since we can scale its output noiselessly to any desired resolution.

While there are many published algorithms for practical curve-fitting of bitmap edges (“autotracing”) [7], none attempts the possibility of recovering the exact mathematical curves underlying a noiseless rasterization such as METAFONT generates. (The typical application tries to fit approximating curves to a noisy scan of an irregular physical object.) In this matter, this author is again tantalizingly close to a curve-fitter that will solve the problem and implement it in robust form. Among other wonderful applications, this would allow conversion of METAFONT shapes to outlines by “mere” curve-fitting.

Other Meta-Design Formats. Besides METAFONT, there are other formats for specifying some degree of meta-design to typefaces, such as the multiple master extension to Type 1. But none of these match the potent ability of METAFONT to express meta-ness in far more sophisticated ways than mere linear interpolation. Linear interpolation may be sufficient for a limited range of variation, such as

stem weight, slant, or even the presence or absence of serifs. But the non-linear and programmatic possibilities of METAFONT provide a much wider range of possible variations; and still more powerful is METAFONT’s ability to stroke and overlap shapes.

On the other hand, some anecdotal experience has resulted in failures when attempting satisfactory METAFONT designs [8]. One lesson from such experience is that a single meta-character is not necessarily able to represent wildly different characteristics, particularly as might vary in letters. In the case of nearly all non-letter math symbols, however, one can expect that the possibilities of variation are restricted enough to permit meta-characterization to a degree sufficient to cover a wide range of text typefaces. It might be necessary to produce differing math meta-characters for serif versus sans-serif typefaces, or other gross variations; indeed this was the effect of many of Knuth’s conditionals in Computer Modern.

The Future: \TeX , and the Web

The future of math publication, whether in \TeX or on the Web, will depend in part on the variety of math fonts available. It would appear inevitable that math fonts will always lag seriously behind text fonts, if creation of quality math fonts necessarily involves manual design. The present tools for meta-font design suffer from a fundamental flaw in that they cannot produce parametric output, only bitmaps. This approach is impractical, if for no other reason than it necessarily involves intractible complications for users, who cannot be expected to deal with the vagaries of bitmapped fonts.

The tasks of defining font encodings, building a symbol inventory, designing meta-math programs, and writing style-switching code are all substantial, yet well-understood. None of those problems will ultimately impede the adoptions of \TeX , HTML, or any other markup language. They involve complicated details which can be managed by the experts and well-hidden from the user.

It is therefore our conclusion that:

- Quality math meta-fonts will be crucial to success in math publishing, because hand-drawn shapes are too costly.
- Implementing quality meta-fonts reduces to two fundamental problems:
 - A language for meta-design, which we believe is superbly extant in the METAFONT language.
 - A processor for that language which can produce non-overlapping outlines. This

requires solving one of two open research problems: topological analysis of stroked, overlapping shapes; and exact curve-fitting of arbitrary-resolution rasterizations.

References

- [1] Hobby, John D. “A METAFONT-like System with PostScript Output.” *TUGboat* **10** (4), pp. 505–512, 1989. See also the software on CTAN.
- [2] Hoenig, Alan. “Hundreds of New Math Fonts with MathKit (version 0.7).” CTAN `fonts/utilities/mathkit`.
- [3] Hoenig, Alan. “The MathInst Package (version 0.8): New Math Fonts for T_EX.” CTAN `fonts/utilities/mathinst`.
- [4] Kinch, Richard J. “MetaFog: converting METAFONT shapes to contours.” *TUGboat* **16** (3), pp. 233–243, 1995. Current version available with TRUET_EX.
- [5] Knuth, Donald E. *The METAFONTbook*. Addison Wesley, Reading MA, 1986.
- [6] Knuth, Donald E. *METAFONT: The Program*, Section 524, “Elliptical Pens”. Addison Wesley, Reading MA, 1986.
- [7] Schneider, Philip J. “An algorithm for automatically fitting digitized curves.” In *Graphics Gems*, Andrew S. Glassner, editor, pp. 612–626 and 797–807. Academic Press, Cambridge MA, 1990. See also the on-line archives at <http://www.acm.org>.
- [8] Siegel, David R. *The Euler Project at Stanford*. Department of Computer Science, Stanford, CA, 1985.
(This “illuminating little booklet” on negative experience attempting to metafont-ize the Euler font at Stanford was cited by Berthold Horn in `news://comp.fonts` ca. Aug 1997.)

Georgian Scripts

Karel Piška

Institute of Physics, Academy of Sciences

180 40 Prague, Czech Republic

piska@fzu.cz, piska@cern.ch

<http://www-hep.fzu.cz/~piska/>

Abstract

Georgian writing is presented by three historical development steps: 1. *asomtavruli* (the earliest ‘capitals’), 2. *nusxa-xucuri* (later ‘minuscule’), and 3. *mxedruli* (modern Georgian).

The standard and headline printed forms of *mxedruli* shapes are based on the Computer Modern Georgian font designed by **Nana Glonty** (1994). The standard *mxedruli* font is extended with several letters that were formerly used for Georgian and some other languages in the Caucasus; a modified definition (co-ordinate transformation) is used in the font for headlines and titles. Fonts for two old historical scripts and a handwriting form of *mxedruli* with numerous letter connections (no ligatures are used in printed forms) are presented here as designed by the author of this paper.

Introduction

Five alphabetic writing systems (and their adaptations) denoting consonants and vowels and written from left to right are used today for modern literary languages, in chronological order of their creation: Greek, Roman (Latin), Armenian, Georgian, and Cyrillic.

The first Georgian alphabet was invented in the 5th century, presumably influenced by the Aramaic script and the Greek alphabet. The earliest inscriptions are written (or graven in stone) in inscriptional capitals, in the ancient Georgian script known as *ასომთავრული*¹ *asomtavruli* ‘capital letter’, ‘majuscule’ (also called *პრეგლოვანი mrglovani* ‘rounded’), and can be found in Palestine and Georgia. The earliest example dating from 430 AD, is in the Georgian monastery in Bethlehem. Its text is presented in Example 3a. Another sample is given in Example 3b.

Another script, known as *ნუსხა-ხუცური nusxa-xucuri* ‘priest (church) minuscule’ or *ნუსხური nusxuri* ‘minuscule’ (also called *კუთხოვანი kutxovani* ‘angular’) appeared in the ninth century.

The modern Georgian script, *მხედრული mxedruli* (from *მხედარი mxedari* ‘warrior’, i.e., secular) started its development in the eleventh century. It is used for writing the modern Georgian literary

language; it has been used also for writing (and transliterating) other Georgian dialects, and other languages in the Caucasus, related and unrelated to Georgian: Mingrelian, other Kartvel languages (without a literary form), Ossetic, Abkhaz.

All three scripts continued in a parallel existence for several centuries. While *mxedruli* prevailed for secular functions and everyday handwriting, the two older scripts, *nusxa-xucuri* and *asomtavruli* (often together called *ხუცური xucuri*; from *ხუცესი xucesi* ‘priest’), continued to be used in religious writing, *nusxa-xucuri* being more convenient for manuscript texts, and *asomtavruli* for initials and titles (see Example 4 with both scripts).

More information about the languages and references to further sources can be found in the book “The World’s Writing Systems” [1]. Letter shapes and samples were taken from [2–5, 7, 8].

Alphabet

Table 1 presents the characters of the Georgian alphabet: the regular and headline forms of *mxedruli*, the two old scripts, phonetic values, two more usual transliteration systems, Greek and Armenian equivalents. None of the Georgian scripts have capitals; more accurately, there are no uppercase/lowercase pairs — *asomtavruli* (‘majuscule’) and *nusxuri* (‘minuscule’) may be assumed to be the names of two “historical” stages of evolution like Roman Square Capitals or Carolingian Minuscule, i.e., two distinct scripts.

¹ A Georgian script used implicit inside the English text is *mxedruli*. The transliteration of the Georgian words follows in *IKE* (in *italic*); see Table 1 for phonetic values to determine how to read it.

Georgian Alphabet Table 1.

m	LETTER			IPA	LETTER NAME		TRANSLITERATION		NUMERAL	EQUIVALENT	
	h	a	n		mn	u	<i>IKE</i>	<i>LC</i>		grk	arm
ა	ⴀ	ⴁ	ⴂ	[a]	ან	AN	a	a	1	Aα	Աա
ბ	ⴃ	ⴄ	ⴅ	[b]	ბან	BAN	b	b	2	Bβ	Բբ
გ	ⴆ	ⴇ	ⴈ	[g]	გან	GAN	g	g	3	Γγ	Գգ
დ	ⴉ	ⴊ	ⴋ	[d]	დონ	DON	d	d	4	Δδ	Դդ
ე	ⴌ	ⴍ	ⴎ	[e]	ენ	EN	e	e	5	Eε	Եე
ვ	ⴏ	ⴐ	ⴑ	[v]	ვინ	VIN	v	v	6		Վվ
ზ	ⴒ	ⴓ	ⴔ	[z]	ზენ	ZEN	z	z	7	Zζ	Զզ
ც	ⴕ	ⴖ	ⴗ	[ej,e]	ცე	HE	ey,ē	ē	8	Hη	Էէ
თ	ⴘ	ⴙ	ⴚ	[t ^h]	თან	TAN, THAN	t	t ^h	9	Θθ	Թթ
ი	ⴛ	ⴜ	ⴝ	[i]	ინ	IN	i	i	10	Ιι	Իի
კ	ⴞ	ⴟ	ⴠ	[k ^h]	კან	KAN	k	k	20	Κκ	Կկ
ლ	ⴡ	ⴢ	ⴣ	[l]	ლას	LAS	l	l	30	Λλ	Լլ
მ	ⴤ	ⴥ	⴦	[m]	მან	MAN	m	m	40	Μμ	Մմ
ნ	ⴧ	⴨	⴩	[n]	ნარ	NAR	n	n	50	Νν	Նն
ო	⴪	⴫	⴬	[j]	ოე	HIE, YE	y, j	y	60		Յյ
პ	ⴭ	⴮	⴯	[o]	ონ	ON	o	o	70	Οο	Օո
ჟ	ⴰ	ⴱ	ⴲ	[p ^h]	პარ	PAR	p, p ^h	p	80	Ππ	Պպ
ჭ	ⴳ	ⴴ	ⴵ	[ʒ]	ჭან	ZHAN, JAN	ž	ž	90		Ժժ
რ	ⴶ	ⴷ	ⴸ	[r,r ^h]	რან	RAE	r	r	100	Ρρ	Րր
ს	ⴹ	ⵀ	ⵁ	[s]	სან	SAN	s	s	200	Σσς	Սս
ტ	ⵂ	ⵃ	ⵄ	[t ^h]	ტარ	TAR	t	t	300	Ττ	Տտ
კ	ⵅ	ⵆ	ⵇ	[wi]	კე	WIE	wi,ü	w	400	Υυ	Է
უ	ⵈ	ⵉ	ⵊ	[u]	უნ	UN	u	u	(400)	ΟΥου	Ուու
ფ	ⵋ	ⵌ	ⵍ	[p ^h]	ფარ	PHAR, FAR	p	p ^h	500	Φφ	Փփ
ქ	ⵍ	ⵎ	ⵏ	[k ^h]	ქან	KHAN	k	k ^h	600	Χχ	Բբ
ღ	ⵐ	ⵑ	ⵒ	[ɣ]	ღან	GHAN, RAN	ğ,ɣ	ğ	700		Ղղ
ყ	ⵓ	ⵔ	ⵕ	[q ^h]	ყარ	QAR, KAR	q,q ^h	q	800		
შ	ⵖ	ⵗ	ⵘ	[ʃ]	შინ	SHIN	š	š	900		Շշ
ჩ	ⵙ	ⵚ	ⵛ	[tʃ]	ჩინ	CHIN	č	č ^h	1000		Չչ
ც	ⵜ	ⵝ	ⵞ	[ts]	ცან	CAN, TSAN	c	c ^h	2000		Յყ
ძ	ⵟ	ⵠ	ⵡ	[dʒ]	ძილ	JIL, DZIL	j,ʒ	ž	3000		Ձა
წ	ⵢ	ⵣ	ⵤ	[ts ^h]	წილ	CIL, TSIL	ç	c	4000		Շծ
ჭ	ⵥ	ⵦ	ⵧ	[tʃ ^h]	ჭარ	CHAR	č	č ^h	5000		Ճճ
ხ	⵨	⵩	⵪	[x]	ხან	XAN, HAN	x	x	6000		Խխ
ჯ	⵫	⵬	⵭	[q]	ჯარ	HAR	q	q ^h	7000		
ჯ	⵮	ⵯ	⵰	[tʃ]	ჯან	JHAN, DJAN	ž,ž ^h	j	8000		ՋՋ
ჰ	⵱	⵲	⵳	[h]	ჰან	HAE	h	h	9000		ჴჴ
მ	⵴	⵵	⵶	[ow]	მე	HOE, OH	ow,ō	ō	10000	Ωω	

^m standard (regular) *mxedruli*, ^h headline form of *mxedruli*, ^a *asomtavruli*, ⁿ *nusxa-xucuri*;
^{mn} letter name in *mxedruli*, ^u in English (the first one is Unicode);
IKE transliteration as presented in *Annual of Ibero-Caucasian Linguistics*,
LC the Library of Congress transliteration;
^{grk} Greek equivalents, ^{arm} Armenian equivalents.

The Georgian alphabet is phonemic, i.e., it follows a one-to-one correspondence between phonemes (sounds) and characters (with only rare exceptions). The alphabetical order agrees with that of the Greek alphabet; letters that are not present in Greek are located at the end.

The alphabet originally contained 38 characters (see Table 1) but only 33 letters after 1860, when five letters (⊗ ē [e(j)], α y [j], ζ w [w(i)], ξ q [q], and θ ო [o(w)]) were dropped. The letter η u [u] was composed as a ligature ο+ζ o+w [o]+[w]. As can be seen in Example 4 (specified by boxes) u was represented by the digraph **⊗ϥ** in *asomtavruli* or the ligature **⊗ϣ** (= ⊗+ϣ) in *nusxa-xucuri*. The symbol **⊗** was introduced later also for o. The sound u was denoted formerly by **⊗ϥ**, then written **⊗ϣ**, and only later **⊗** became u corresponding to Table 1.

Additional letters are designed for other languages than Georgian (only for printed *mxdedruli*; the Cyrillic equivalents are shown after slashes): φ/ϕ f [f], ჯ/Ɑ shwa [ə], ს/æ æ [æ] for Ossetic and/or Abkhaz [5], Ɔ '(glottal stop) [ʔ], Ɔ' uo [u] for Mingrelian (Megrelian) [6].

Alphabetic numeral notation was used formerly (similarly to other alphabets). Arabic numerals as well as Roman numerals and typical European punctuation are used today.

Two Unicode names are corrected here: ZHAN and KHAN (not ZHAR, KHAR).

Fonts

METAFONT sources for printed forms of *mxdedruli* and *asomtavruli* were designed using the “Computer Modern technology” (cmbase.mf and commands like **penpos**, **filldraw**, etc.) with modifications and extensions similar to those used for defining “DC/EC fonts” (parameter files as tables).

Table 2. Additional letters for <i>mxdedruli</i> (not present in original font designed by Nana Glonty, 1994)	
⊗ⱭⱮⱯ	Old Georgian
ⱭⱮⱯ ⱭⱮ	Ossetic, Abkhaz, Mingrelian

There is one variant for *asomtavruli* (not listed in Table 1): The letter **Ɑ** b has a variant **Ɑ**.

Nusxa-xucuri was created in a simple way and is very plain. There are tens of letter shape variants in manuscripts and I decided that the simple font design can be “topologically” similar to many of them.

Handwritten form of *mxdedruli*

The handwritten form of *mxdedruli* (see Table 3) is also drawn by a simple pen. On the other hand, it contains numerous letter connections (see Example 2). Not all letters in a word should be joined, so only parts of words are continuous. Five obsolete letters are omitted.

Table 3. Handwritten form of <i>mxdedruli</i>		
Long and short variants (may not be obvious to everyone)		
Printed Form	Handwriting Long	Short
Ɑ		
Ɱ		
Ɐ		
Ɒ		

Examples

The next few pages show several short examples of all Georgian scripts.

Example 1. Printed *mædruli*

აკაკი წინათელი
სულიკო

საყვარლის საფლავს ვეძებდი,
ვერ ვნახე... დაკარგულიყო!...
გულამოსკენილი ვჩიოდი:
„სადა ხარ, ჩემო სულიკო!“

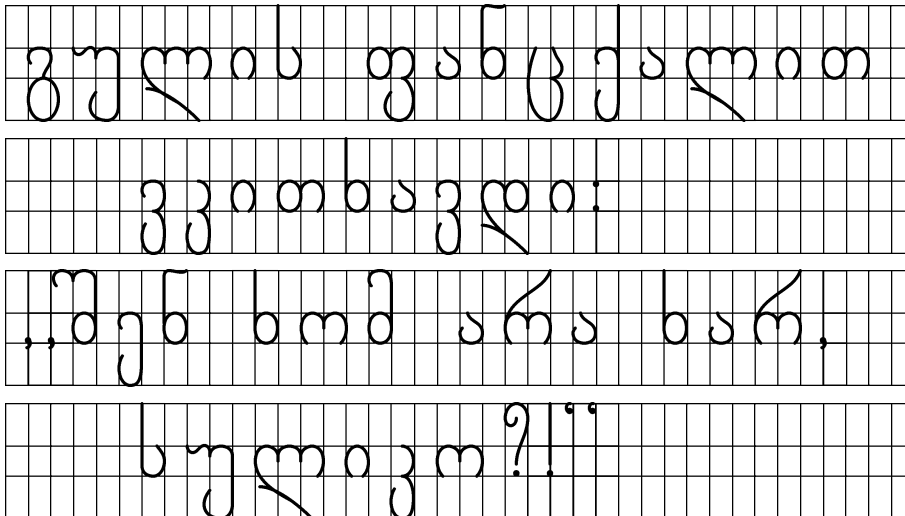
Headline form (the same
height and the zero depth)

and regular form of
printed *mædruli* მხედრული.

Handwritten forms taught in Georgian schools

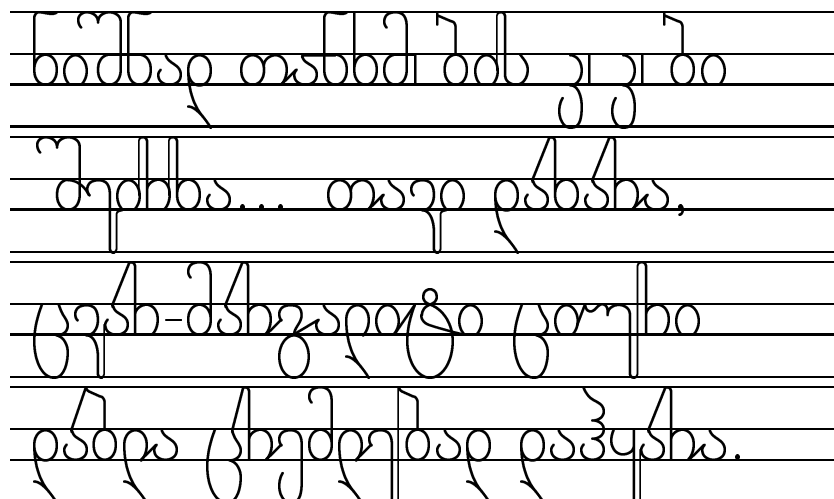
Exercise books for the beginning years of a Georgian school are preprinted with special grids, especially the books for the first year. The second year needs only four horizontal lines. These represent “visualized” imaginary upper-, middle-, lower-, and baseline. Pupils (or students) of Georgian handwriting learn to fill the characters into boxes bounded by frames or between the lines. Starting in the third year, pages with one baseline are used; letter heights and depths may be more free.

Example 2.a Hand-written *mædruli*



პირველი
კლასი

(The first
year)



მეორე
კლასი

(The second
year)

Example 4. *Nusxa-xucuri*

ოც მუ უთარეხმენ მუხე
 არ ყუყორაჲ ონ ოსე მბ
 ერეოთერო ოთ ომბე
 სე მთესე სრეს სეესე
 ნიყუთე ოც სეაუყუთე
 თერთე ჩინთე ოც სთთ
 ე ჩინთესე ოც ემეს
 ყრინ ეას ყეძესე ო
 თორთბესე ონესესე **ოჲ**
 თთე ყთთეა თქთო ოთთე
 მბაბო **ჲ**ესესე
 მუა ორ ანა უარინე ოყესე
 აყესე ყრინ მეს ონესე
 ყრინ სთთე ჩინთე ო
 ბორთთე ოც აყოყესე
 აბ უაყეძესე ოყესე
 ორს მბაბოყესესე მბოჲ
 ოყესესესე
ო ოესე სთო ჩინთე სესე
 ბორსესე თაყესესე
 ორთესესე ოც ორთესე
 ყრინ ჩინთე ყესესესე
ო ოესე ონესე ონესე
 ონესე ონესე ონესე
 ონესე ონესე ონესე

სინური მრავალთავი, 864 წ., ანდერჯი, 274გ, *sinuri mravaltavi*, 864 წ., *anderji*, 274გ, [2], p. 83.

The text in *nusxa-xucuri* has initials and appendix in *asomtavruli*. The digraph **ოჲ** or the ligature **ჲ** denotes the sound [u] (ჲ *u*). Upper horizontal bars mark conventional abbreviations of specific words.

Conclusion

An extension of the modern Georgian font and other Georgian fonts (including the old scripts) were created in the first release.

Acknowledgements

I would like to thank Nana Glonty for her excellent font design of the modern Georgian script and all authors of free fonts (listed in the references) for possibility to print the Greek and Armenian equivalents, IPA values and other texts.

References

- [1] The World’s Writing Systems. Edited by P.T. Daniels and W. Bright. Oxford University Press, New York – Oxford, 1996.
- [2] ი. აბულაძე. ქართული წერის ნიმუშები. მეცნიერება, თბილისი, 1973. [I. Abuladze. Samples of Georgian Script. Mecniereba, Tbilisi, 1973.]
- [3] Г.И.Цибыхашвили. Самоучитель грузинского языка (элементарный курс). Изд. 4-е, Тбилиси 1981.
- [4] Ю.Н. Марр и И.В. Мегрелидзе. Пособие для авторов и наборщиков восточных шрифтов. Мецниереба, Тбилиси, 1984.
- [5] Р.С. Гиляревский, В.С. Гривнин, ‘Определитель языков мира по письменностям’, Издательство восточной литературы, Москва, 1960.
- [6] A. Kharchilava, personal message.
- [7] International Organization for Standardization. Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane. ISO/IEC 10646-1 : 1993, (First edition, 1993-05-01), Geneva, 1993. (Unicode version 1.0)
- [8] <http://charts.unicode.org/charts.html> Unicode 2.1 Character Charts, 1998.
- [9] /CTAN/fonts/greek/cb/ Greek font – C. Beccari, 1997.
- [10] /CTAN/fonts/armenian/ Armenian font – S. Dachian, V. Hakobian, 1997.
- [11] /CTAN/language/cyrillic/ WN Cyrillic font – B. Beeton, T. Ridgeway, 1987–1995.
- [12] /CTAN/fonts/tipa/ phonetic font – F. Rei, 1996.

Generating Type 1 Fonts from METAFONT Sources

Taco Hoekwater
Kluwer Academic Publishers
Dordrecht
taco.hoekwater@wkap.nl

Abstract

This article makes a comparison between bitmapped and vector fonts, and presents some of the problems I encountered when I tried to convert METAFONT sources into PostScript Type 1 fonts.

The second part of this article will focus more closely on some of the problems that I faced while trying to convert METAFONTS into PostScript Type 1 fonts, but first some explanation is in order as to why one might want to do this conversion, and precisely what this conversion entails. These topics are the subjects of the first couple of paragraphs.

What are METAFONT Fonts?

How characters are created

I'll assume the reader knows the following: every T_EX distribution has a program called METAFONT, that compiles font sources more or less the same way that the T_EX program compiles text sources. A major difference between the two programs is that METAFONT produces *device-dependent* output (called `pk` files), whereas T_EX produces *device-independent* output (also known as `dvi` files).

Let's look into the font sources that METAFONT uses, and see what kind of information they contain. These are ordinary ASCII files just like T_EX sources, so it was easy to insert a listing of one of these files. The file that contains the METAFONT logo font (`logo10.mf`) suits our purpose quite well, since it is a rather simple font that probably everybody has available:

```
font_size:= 10pt# ;
ht#      := 6pt#  ;
xgap#    := 0.6pt# ;
u#       := 4/9pt# ;
s#       := 0     ;
o#       := 1/9pt# ;
px#      := 2/3pt# ;
input logo
bye
```

What do we see here? First there are a bunch of assignments (the lines that contain `:=`), then there is an `input` (this command functions the same way as T_EX's `\input`, so it will start reading the file `logo.mf` next), and finally the last command is `bye`.

The file `logo.mf` contains the actual commands to create the characters. It helps to run METAFONT

now to see what is going on. Just type the following to a system command prompt:

```
mf logo10
```

Probably you didn't have to worry about where the `logo10.mf` file is on your hard disk, since most METAFONT implementations can do recursive directory searches just like T_EX.

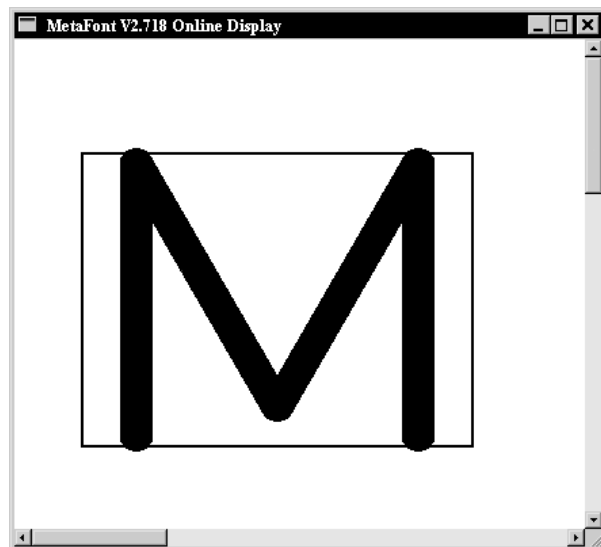


Figure 1 METAFONT output window for modeless files

You should have seen a window popping up that shows characters as they are being created (like the one in figure 1, but it might look a little different on your system). Also, there should have been some terminal output, like this:

This is METAFONT, Version 2.718 (Web2c 7.2beta7) (logo10.mf (logo.mf [77] [69] [84] [65] [70] [80] [83] [79] [78]))
 Output written on logo10.2602gf (9 characters, 98 80 bytes).
 Transcript written on logo10.log.

The numbers you see are the positions of the characters in the font. For the METAFONT logo font, these are the positions of the used characters in the ASCII table: M, E, T, A, F, P, S, O, N. As you can see, they can appear in any order within the source files.

The file METAFONT has written is not precisely the same as the output on your screen, instead it looks like figure 2. Not really usable when it comes to typesetting text, but it contains some pretty valuable information nevertheless.

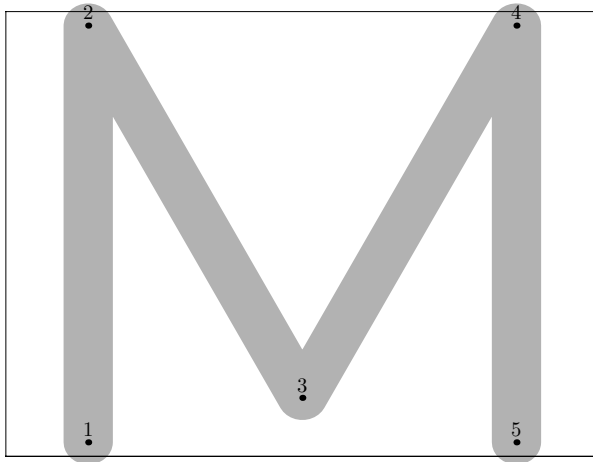


Figure 2 Metafont output file for modeless files

Have a look at the `logo.mf` file if you are interested in the nitty-gritty details. I will use only a small portion of that file to make some remarks about METAFONT. First, here is the edited program text I will use to explain things. (This is *no longer* valid METAFONT input, so don't start keying it in):

```
mode_setup;
define_pixels(s,u);

ygap#:= (ht#/13.5u#)*xgap#;
define_whole_pixels(xgap);
define_whole_vertical_pixels(ygap);

py#:= .9px#;
define_blacker_pixels(px,py);
pickup pencircle xscaled px yscaled py;
logo_pen:=savepen;
```

```
leftstemloc#:=2.5u#+s#;
define_good_x_pixels(leftstemloc);

beginlogochar("M",18);
x1 = x2 = leftstemloc;
x4 = x5 = w - x1;
x3      = w - x3;
y1      = y5;
y2      = y4;
bot y1  = 0;
top y2  = h;
y3      = ygap;
draw z1--z2--z3--z4--z5;
labels(1,2,3,4,5);
endchar;
```

Let us first look at the line that begins with `beginlogochar`, because this is where the real work is done. This portion of the source defines the letter ‘M’ in the font. What we see here is that characters are specified by first setting up a bunch of equations (the lines that have equal signs in them), followed by a `draw` command that connects those points, actually drawing the character.

We won't go deeply into METAFONT syntax, but it is vital to understand the following: a point is defined as a pair of x and y coordinates. In METAFONT syntax, points are sequentially numbered per character, starting from 1. `z1` is the notation for point 1. Notations like `x1` and `y2` specify the x -component of point 1 and the y -component of point 2, respectively.

`beginlogochar` says that the ‘M’ is precisely $18u$ (units) wide, and `logo10.mf` has set up one u to be $4/9$ pt, so the actual character is $4/9 \times 18$ pt = 8pt wide.

\TeX and METAFONT have the same author, and it shows: METAFONT can do macros just as easily as \TeX can. Macros can have arguments, define other macros and assign values to things, just like in \TeX . `beginlogochar` is in fact one of those macros, and it assigns some pretty important values when it gets expanded by METAFONT. For one, it defines w to be the width we calculated above, and it defines h to be the height of the character (calculated from `ht#` in `logo10.mf`, which equals 6pt).

From the values of u and s , it now follows that `leftstemloc` equals $(2.5 \times 4/9) + 0 = 10/9$ pt. The last value we have left is `ygap` = $(ht/13.5u) * xgap$. `ht` and `xgap` have been given in the ‘driver’ file `logo10.mf`, and after some small calculations `ygap` becomes $(6\text{pt} \div (13.5 \times 4/9\text{pt} \times 0.6\text{pt})) = 0.6\text{pt}$.

It should now be easy to come to the conclusion that the equations fix the precise x, y locations of the

five points that denote the character ‘M’, albeit in a slightly indirect manner. If we fill in the values we derived above, we get the following:

```
x1 = x2 = 10/9pt;
x4 = x5 = 8pt - x1;
x3      = 8pt - x3;
y1      = y5;
y2      = y4;
bot y1  = 0pt;
top y2  = 6pt;
y3      = 0.6pt;
```

After METAFONT has calculated these equalities for us, and with some minor reshuffling of the input, we get the following end-result:

```
x1 = 10/9pt ; bot y1 = 0pt ;
x2 = 10/9pt ; top y2 = 6pt ;
x3 = 4pt    ; y3 = 0.6pt;
x4 = 62/9pt ; top y4 = 6pt ;
x5 = 62/9pt ; bot y5 = 0pt ;
```

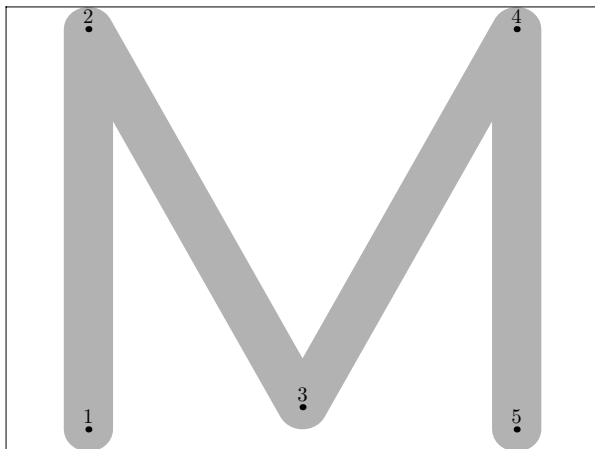


Figure 3 Metafont output file for modeless files, hand-calculated version

We could have typed this in right away, and METAFONT would have been just as happy. The end result would have been the same, as can be seen in figure 3.¹ But why do the calculation yourself if the machine can do it for you?

METAFONT’s ability to do the needed calculations all by itself is one of its most important strong points. Combined with macros and separate input files, it becomes possible to use various fonts with the same shared sources. In such a ‘font’, the only file that is different between various versions of the font is the ‘driver’ file, that assigns different values to the same parameters. METAFONT’s calculations will have different results, so that some of points will end up in slightly different locations. The resulting

font will be similar in style but may still differ in lots of ways.

Creating a full font

Usually, fonts are not just a bunch of characters. There also is some other metric information included in almost every font. The final section of our example file (at the end, after all the characters have been defined) contains the following lines:

```
ligtable "T": "A" kern -.5u#;
ligtable "F": "O" kern -u#;
ligtable "P": "O" kern u#;
```

```
font_quad:=18u#+2s#;
font_normal_space:=6u#+2s#;
font_normal_stretch:=3u#;
font_normal_shrink:=2u#;
font_identifier:="MFLOGO" ;
font_coding_scheme:="AEFMNOPST only";
```

The first three lines belong to the ‘ligature table’. Usually it will contain both real ligatures and the kerning information for the font, but because this is a very simple font, there are only three really simple kerning pairs.

The next lines define T_EX’s `\fontdimen` values: how wide a space will be and how much it can stretch and shrink, and some other information that will appear in the created font but is generally not used by programs.

Dealing with device dependence

Now let’s have a look at the device dependent calculations that METAFONT does. Here is the relevant portion of the example again:

```
mode_setup;
define_pixels(s,u);

ygap#:=(ht#/13 5u#)*xgap#;
define_whole_pixels(xgap);
define_whole_vertical_pixels(ygap);
py#:= 9px#;
define_blacker_pixels(px,py);
```

There are, in fact, two kinds of device dependence that need to be dealt with. The `mode_setup` line takes care of the first kind of device dependence: the effects that the actual hardware of the printing engine can have on the printed font.

The most obvious difference between any two printing devices is of course the resolution, but there

¹ Actually, figure 3 is not completely identical to figure 2, because in my example I cheated with the calculations a bit to keep the explanation simple.

are other problems as well. Since we prefer our output to look as close to our intended font as possible, usually a certain amount of correction is needed based on (i.e.) whether the device is going to be an inkjet printer or a laser typesetter.

`mode_setup` cannot do this all by itself, and this is why you usually have to specify somewhere what printer you are using. Programs like `dvips` will call METAFONT with a command like:

```
mf \mode=ljfour; mag=1; input logo10
```

If we forget about that first backslash, we can see that there are two assignments and one `input` command on this line. The second assignment differs from 1 when a font is called within \TeX using a command like

```
\font\logohuge = logo10 at 20pt
```

In that case, the assignment would be `mag=2`. The other assignment is far more interesting. METAFONT usually starts with a ‘format’ file similar to the `fmt` files \TeX uses, and somewhere in the sources for those format files there are some definitions like this:

```
mode_def cx =
  mode_param (pixels_per_inch, 300);
  mode_param (blacker, 0);
  mode_param (fillin, .2);
  mode_param (o_correction, 6);
  mode_common_setup_;
enddef;
```



Figure 4 An example of two different imaging models.

All parameters besides `pixels_per_inch` are a little too technical to explain in detail in a short article like this one, but figure 4 tries to explain that these values really do depend on the printing engine. The drawing on the left shows a more or less standard inkjet, that shoots dots of (black) ink on the paper. The right drawing shows a (hypothetical) printing device with a radically different approach. This machine pours light on a photographic film through a raster, creating a negative image. There are still round dots, but they are inverted! It is easy to imagine that this radically different technique can

have quite an impact on the resulting image.

One effect that is very easy to see from the (admittedly very badly drawn) figure is that the inside corners in the right drawing are a lot blacker than in the left one. This sort of thing happens all the time in real life printing, but it often goes unnoticed because people tend to have only one printer.

The second device dependency is not really related to printers at all, but is caused simply by the fact that METAFONT outputs a pixel bitmap. Although METAFONT does its calculations with a very high accuracy, this does not help at all if there are simply not enough pixels to display the character. The commands that look like `define_xxx_pixels` take care of this kind of dependency, whose effects can be seen in figure 5.



Figure 5 The character on the right has been created with all the `define_xxx_pixels` commands removed from the source.

The sub-optimal distribution of pixels in this example is caused by the underlying pixel grid that can not be changed.

What are Type 1 fonts?

How PostScript fonts are created

PostScript Type 1 fonts are quite different from METAFONT fonts. Usually, Type 1 fonts are created in a wysiwyg environment with a drawing program that is only suited for the creation of fonts. Figure 6 shows the program I usually use.

The graphical user interface nicely shields the designer from what is happening behind the scenes, so we need to look into the generated files themselves if we want to get more information. On Windows and Unix systems, the actual fonts are saved in a binary file with the extension `pfb` (short for PostScript Font Binary), and the metric information in an ascii file with extension `afm` (short for Adobe Font Metrics).

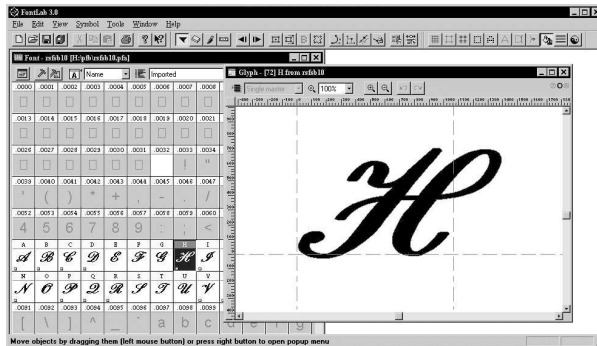


Figure 6 An interactive font editor: Fontlab version 3

What a Type 1 font looks like²

The binary representation of a Type 1 font is just a compressed version of the non-compressed ascii format, with extension pfa. So we need a program that will do the decompression for us. One of the programs that can do this is `T1ascii` from the `T1Utils` package. But running this programs leaves us with a hexadecimal encrypted file. In the early days, the encryption key was a trade secret of Adobe Incorporated. This key is now freely available, but the file format still reflects the past. Yet another program from the `T1Utils` can convert this form to real human-readable PostScript: `T1disasm`. Now we can look at the generated PostScript file to see how the ‘M’ is defined in Type 1 format:

```
/M {
  78 800 hsbw
  611 -20 hstem
  -11 21 hstem
  0 66 vstem
  578 66 vstem
  581 595 rmoveto
  -259 -450 rlineto
  -259 450 rlineto
  -6 9 -12 7 -12 0 rrcurveto
  -16 -17 -12 -17 hvcurveto
  -563 vlineto
  -17 15 -13 18 vhcurveto
  19 14 13 17 hvcurveto
  439 vlineto
  76 -131 75 -131 75 -131 rrcurveto
  5 -10 12 -6 13 0 rrcurveto
  14 0 8 8 8 8 rrcurveto
  75 131 75 131 76 131 rrcurveto
  -439 vlineto
  -17 14 -13 19 vhcurveto
  18 15 13 17 hvcurveto
  562 vlineto
```

```
17 -17 13 -16 vhcurveto
-12 0 -12 -5 -6 -11 rrcurveto
closepath
endchar
} ND
```

The code looks enough like normal PostScript to recognize it at first glance, but the commands themselves are not the same ones you would use in everyday graphics. The PostScript language uses reverse Polish notation for its commands, so you should read backwards, starting at the end of the line. `581 595 rmoveto` means ‘move to the point with coordinates (581,595)’.

All values are given in a coordinate system that maps 1000 units to one `em`. The nullpoint lies at the lower left corner. When one uses a PostScript font in a PostScript language program, the coordinate system is initially scaled in a way such that 1000 units equal precisely 1 `bp`. The values used to describe points and intermediate values can be negative, but never partial. This need for discrete values can be a major problem when converting METAFONT fonts, as we will see later on.

Now let’s have a short look at the used commands. The command `hsbw` sets up the width information for this character (the first number is the left sidebearing distance, the second number the advance width). The commands that end in `stem` are used by the hinting system. The whole collection of commands that look like `xlineto` and `xxcurveto` are shortcuts for the ordinary PostScript commands `lineto` and `curveto`: these draw the actual outline. All of these drawing commands are always relative to the ‘current point’. The last couple of commands end the character: `closepath` to close the defined

² This section is loosely borrowed from Erik-Jan Vens’ article “Incorporating PostScript fonts in T_EX”, EuroT_EX proceedings 1992, pp. 173–181.

path (like METAFONT's `cycle`) and `endchar` to do the actual drawing. ND functions as `def`: it defines the command 'M' (from the first line) to mean 'do everything between the braces' (remember this is reverse Polish notation).

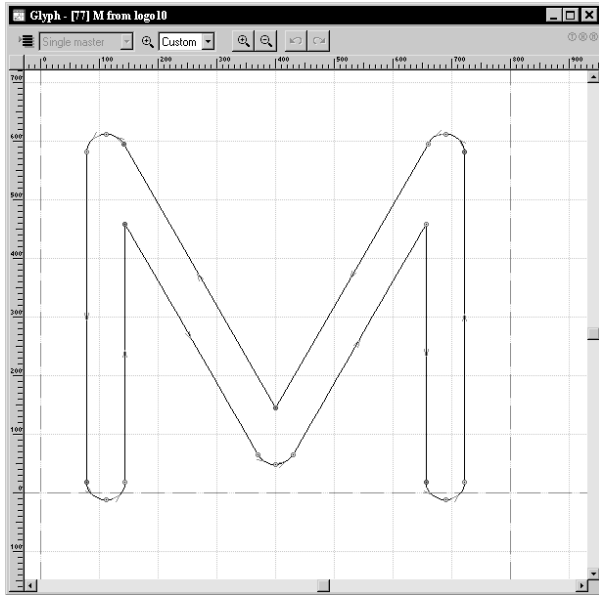


Figure 7 How the character's path is drawn.

At first sight it is a little surprising to see that the PostScript representation is rather a lot longer than the METAFONT version. This is caused by another limitation of Type 1 format: every character *has* to define an outlined path that is filled by `endchar`. Thanks to this limitation, we cannot use four stroked lines to draw the 'M' the way we did in METAFONT, but instead are forced to trace the borders of filled shape.

Dealing with device-dependencies

Adobe's Type 1 format does not supply a means of dealing with device differences directly, like METAFONT's `define_good_pixels`. But of course there has to be some means of making sure that a font looks reasonable on low-resolution devices, and this is handled by a system called 'hints'. The responsible commands are separated into two different levels: there are 'font-level' hints and 'character-level' hints. Font-level hints take care of three things:

1. Alignment zones
2. Standard stem widths
3. Extra information to control the hinting

The relevant portion of the font-file looks like this:

```
/BlueValues [ -12 0 600 611 ] ND
/BlueScale 0.04379 def
/BlueShift 7 def
/BlueFuzz 1 def
/MinFeature { 16 16 } ND
/StdHW [ 60 ] ND
/StdVW [ 66 ] ND
/ForceBold false def
```

Alignment zones. First off, alignment zones are defined by the array called `/BlueValues`. The values in the array define vertical zones by specifying two y coordinates for each zone. In this case, there are only the two areas between $[-12, 0]$ and $[600, 611]$, but there may be more entries.

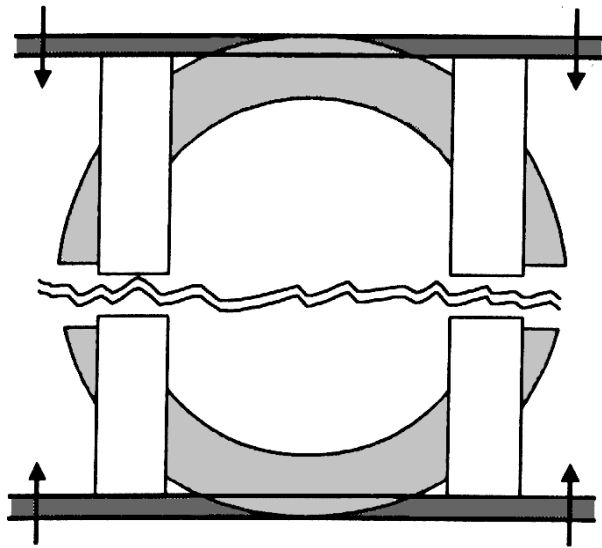


Figure 8 An example of overshoot-suppression in PostScript: the top and bottom of the 'O' are adjusted so that the character becomes just as high as the 'H'. (Figure borrowed from the Fontlab Manual.)

The first entry in the array defines an area in which the y -coordinates of points (that lie within this area) are changed into the highest (second) number. For the following entry, the y -coordinate is changed into the lowest (first) number. Together, these two areas allow characters like the 'O' to be rendered at low resolution without sticking out unacceptably below the baseline if compared to characters like the 'H' (see figure 8).

Standard stem widths. Quite often, a vertical or horizontal line in a font will be just a little bit too large for one device pixel but not large enough for two pixels. Depending on the underlying pixel grid, the line may consequently be rendered as either one

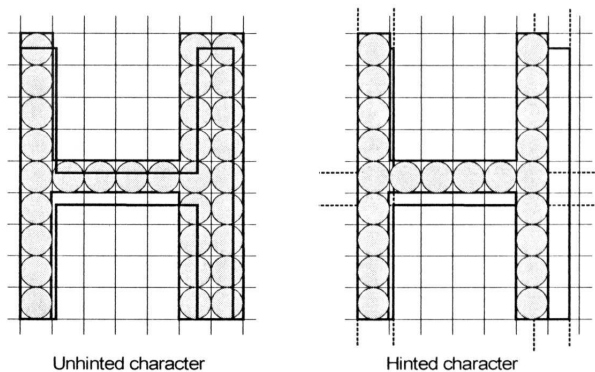


Figure 9 The desired result. (This figure is also borrowed from the Fontlab Manual.)

or two pixels.

In these problematic characters, we want to make sure at least that verticals and horizontals that are intended to have the same width throughout the font use the same number of pixels. This is done by pre-defining the widths that are supposed to be identical. The commands that pass this information to the renderer are `StdVW` and `StdHW`. The effect that a correct setting of these values has on the rendering of the font can be seen in figure 9 (Like all hinting information, these values are ignored if the stem widths are larger than three device pixels—approximating 1200 dpi for the average font. As a result, output at 1200 dpi on a new device sometimes looks inferior to the 600 dpi version for the trained eye.).

Extra information to control the hinting.

There are some extra commands in the example that we haven't covered yet: the three `Bluexxxx` commands define (amongst other things) the pointsize below which overshoot suppression is turned on, and a fuzzy correction on the values of the alignment zones. `ForceBold` is used with bold fonts to make sure that they will stay at least two pixels wide at low resolutions (otherwise they would look identical to the non-bold version at small sizes).

The character-level hints are handled by the commands from the top of the listing given previously:

```
611 -20 hstem
-11 21 hstem
0 66 vstem
578 66 vstem
```

These define horizontal and vertical stem zones. The first number says at which coordinate to start, the second number the width to use from there. In this case (remember this is an 'M') there are two vertical

stems, and two 'ghost' horizontal stems. Figure 10 shows the graphical representation of this character in the font editor.

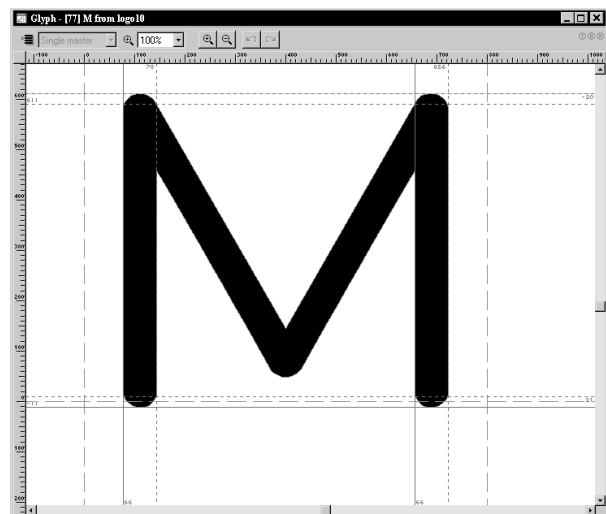


Figure 10 The character 'M' from the META-FONT logo font, with PostScript hints added

The 'ghost' stems are inserted because without them the overshoot suppression wouldn't work.

Why we want to convert to Type 1

Now that we have looked briefly into both formats, it is obvious that conversion from METAFONT input syntax to PostScript definitions is not going to be easy. METAFONT is apparently a lot smarter than the Type 1 interpreter, much better suited to handle device dependencies, and more accurate.

So, why bother at all? For practical reasons, of course. The most important incentive is the on-screen display of generated PDF files. Adobe's Reader is very bad at displaying bitmapped fonts, so files with only Type 1 fonts look a lot better. As it is, there are quite a few METAFONT fonts that don't (yet) have a Type 1 counterpart, so necessarily lots of T_EX-generated PDF files use bitmaps.

There also is another interesting motive: designing high quality fonts in METAFONT syntax is a lot easier than creating Type 1 fonts of the same quality in an interactive editor (not to mention the fact that interactive programs usually crash at every second mouse click).

Tasks to be handled by the conversion process

Various things need to be taken care of by the conversion, but the three major parts are:

1. Resolving the equations in the METAFONT sources.
2. Converting stroked paths into outlined paths.
3. Insertion of Type 1 style hinting information.

Resolving the equations in the METAFONT sources

The first item is easy to do with an already existing program: METAPOST. METAPOST is a program by John Hobby (co-author of METAFONT) that accepts METAPOST input syntax and outputs an Encapsulated PostScript picture. For example, running METAPOST on the logo fonts (using precisely the same syntax as for METAFONT) gives the following output:

```
%!PS
%%BoundingBox: 0 -1 8 7
%%Creator: MetaPost
%%CreationDate: 1998.05.10:1535
%%Pages: 1
%%EndProlog
%%Page: 1 1
0.66418 0 dtransform exch truncate
    exch idtransform pop setlinewidth
[] 0 setdash
1 setlinecap
1 setlinejoin
10 setmiterlimit
gsave
newpath
1.10696 0.18819 moveto
1.10696 5.78938 lineto
3.98503 0.78595 lineto
6.8631 5.78938 lineto
6.8631 0.18819 lineto
1 0.9 scale
stroke
grestore
showpage
%%EOF
```

The PostScript code contained in this file is not that hard. The first few lines are just comments. The two lines that end with `setlinewidth` do nothing except setting the line width for strokes. It looks complex, but the code is always the same, the only things in these two lines that ever change are the

two numbers.

The next lines set up some values of the PostScript graphics state that do not always have a predefined value (this is just a security measure). These lines also never change. `newpath` is the first command that is interesting: starting from here the character is defined. Indeed, there is only one `moveto`, followed by four straight lines, and finally a `stroke`.

It would be a little bit easier if the calculated values were given in units of a thousand per *em*, and this can be done by inserting a different `mode_def`. Basically, we ask METAPOST to generate a normal font file, but at a magnification of 100.375. This gives us an end result in PostScript big points, and the generated character will now look like this (some comments and irrelevant lines stripped):

```
%!PS
%%BoundingBox: 77 -12 723 612
66.66722 0 dtransform exch truncate
    exch idtransform pop setlinewidth
gsave newpath 111.1115 18.88889 moveto
111.1115 581.11142 lineto
399.99867 78.88953 lineto
688.88585 581.11142 lineto
688.88585 18.88889 lineto
1 0.90001 scale stroke grestore
showpage
%%EOF
```

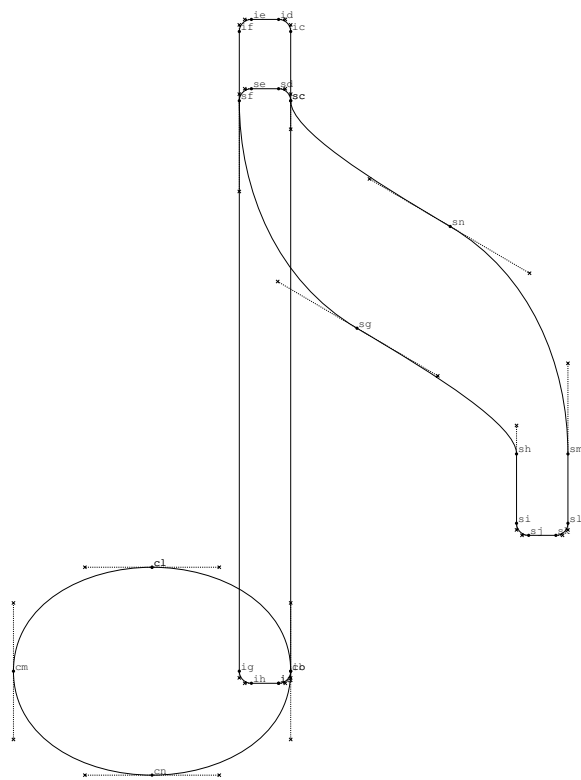
Good. This is starting to look like something we could use. Ideally, we would prefer an output in rounded numbers, but that is not possible.

Converting stroked paths into outlined paths

Mr. Kinch (author of True \TeX) has written a program called Metafog that converts METAPOST output as in the example above into the format required by the Type 1 specifications. At the moment, the program is only available as an optional extra with True \TeX , but correspondence with Mr. Kinch indicate that it is very likely that this program will soon be available separately.

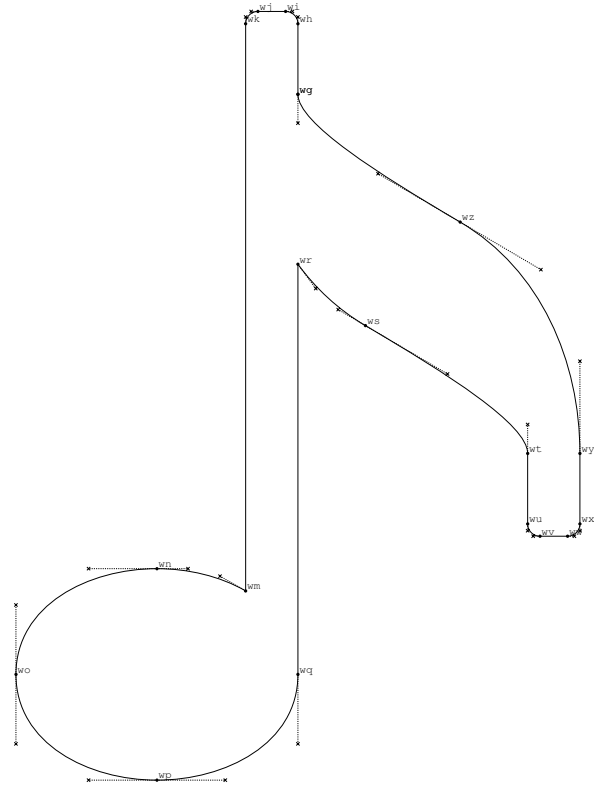
Metafog reads the METAPOST EPS file, and converts this into another EPS file. It also displays some debugging information about the character on the terminal:

```
interp: "scale" implies elliptical pen, \
                                         66.7 x 6 0.0
o: scaffolded TRUE
reduce: reducing shape 1 of 2
reduce: reducing shape 2 of 2
```



Page 1--Initial Input Contour

Figure 11 Metafog output file, page 1



Page 2--Final Result Contour

Figure 12 Metafog output file, page 2

```

duplicate: scaffolded
try_point: 0.01 value scaffold
reduce: reducing shape 3 of 2
.....
reduce: reducing shape 5 of 4
Plotting page 1 (Initial Input Contour) \
    ... done plotting.
reduce: reducing shape 1 of 1
reduce: reducing shape 2 of 1
Plotting page 2 (Final Result Contour) \
    ... done plotting.
Total knots used: 598 (a--wz), ~ 29% \
    indexable capacity
    
```

The created files are pretty large, too large to include literally. This is because the file serves two purposes: it is the input format for another program (Makefont, also by Kinch) and it also shows the work that has been done by Metafog. The first page shows the result, the second page the initial input as Metafog saw it. (The output of one of these files is shown in figure 11 and figure 12.)

All we have to do is run Metafog on all characters in the font. If everything went correctly, the next step in the process is running the Makefont

program. But this is not always the case. Metafog has its flaws, and it is especially bad at handling complex characters. One of those trickier characters is given in figure 13.

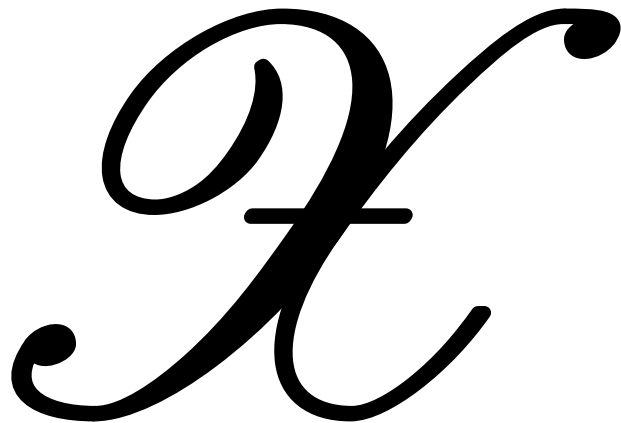


Figure 13 The character 'X' from Ralph Smith's Formal Script

In order to handle cases like this gracefully, Metafog has a special startup option that gives a half-way result: it cuts the supplied shape in pieces, but it does not try to remove parts that are not needed.

There is yet another program in the Metafog suite that helps for the problematic characters.

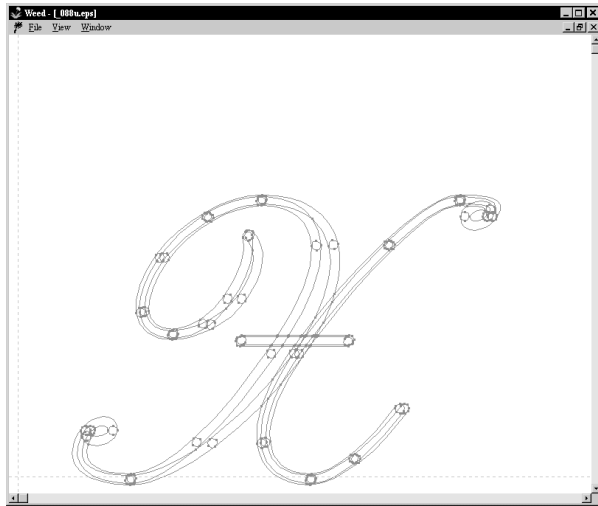


Figure 14 Screenshot of the weeder's window

This program is called the 'weeder' (figure 14). It is an interactive program that reads the half-way result Metafog created. The human operator now has to select the partial lines that are supposed to belong to the shape, and the weeder will write a finished file for use by Makefont (just like Metafog itself would have done if things had gone right the first time).

For some fonts, one has to do almost every character 'by hand', for other fonts none at all. The point of view the machine has on what precisely denotes a complicated character can be rather unexpected: sometimes a character can look very simple to you but be almost impossible to process by Metafog (usually characters that use `draw` and `fill` commands that intersect somewhere). And the other way around also happens: large portions of the `nash14` (arabic) font looked exceedingly complex to me, but were in fact handled by Metafog without any problems.

Either way, eventually there will be EPS files available for all characters in the font. Makefont combines all of the separate files into one PostScript file, and the last step of the actual conversion process is running the T1Utils to get a binary representation that can be fed into a commercial font editor.

Insertion of Type 1 style hinting information

The `pfb` file created at the end of step two still has a couple of major flaws that need to be fixed. First and foremost among these: there are absolutely *no* Type 1 hints included. There were hints in the

original METAFONT sources, but these are ignored by METAPOST, and subsequent portions of the conversion do not have access to them. This is the major reason for the need of a commercial font editor. Type 1 hinting is too complicated a process to rely on any non-interactive program to make the right choices.

Another thing that must be checked, especially for symbolic fonts, is the turning direction of the subpaths. In PostScript, whether a path will be black or white depends on how the path turns: clockwise or counterclockwise. Metafog sometimes gets confused, and outputs a character in which two concentric circles both turn leftward (like in an 'O' or the diameter symbol from the Waldi Symbol font). In those cases, the character will be completely filled, which is of course wrong.

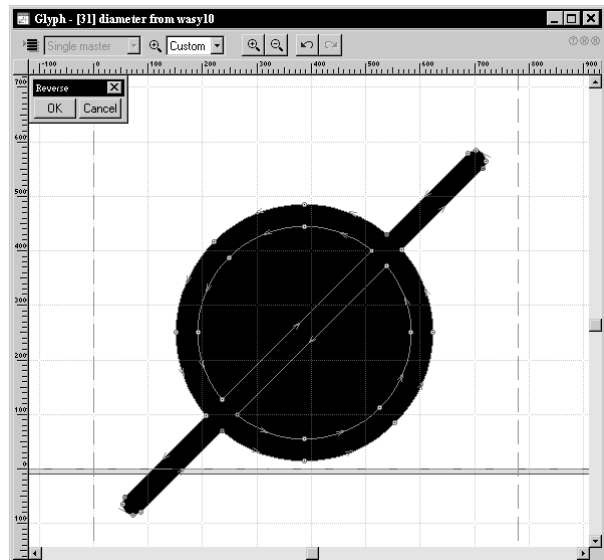


Figure 15 Paths that turn the wrong way

The absolutely final step (so far I've needed to do this for every commercial font program I could find) is disassembling the `pfb` file, running a perl script to fix some incompatibilities/bugs in the used font editor and to insert a couple of workarounds for bugs in software that uses the font, and reassembling.

What I have done already and future plans

So far, I have converted four METAFONT fonts that I needed myself. The files that are now available from CTAN are:

logo wasy2 stmary rsfs

All of these files reside in a subdirectory of the METAFONT sources, named `ps-type1/hoekwater`. Each directory also contains a README file that gives some detailed information about the font in question and its copyrights.

Please take note of the fact that I *only* want to give support for problems that are *intrinsically* related to the `.pfb` files themselves. I don't have enough spare time to help people with problems related to the integration of the fonts into their T_EX distributions. If somebody wants to volunteer for this job, please let me know and I will add you to the readme.

I plan to add other fonts in the near future (some of which may have been uploaded already by the time you read this). The original announcement of the availability of the files that are now on CTAN almost immediately resulted in a doubling of the length of my wish list.

The following fonts are *TODO* and will definitely be done before the summer:

- At least the most important fonts that are needed by the `wsuipa` package: `tipaxx` and `xipaxx`
- The Nash font that is used by ArabT_EX (Klaus Lagally says that he needs to fix and update the METAFONT sources first).
- The Blackboard Bold font (actually, everything that is needed for the new math font encoding will be available in Type 1 before the end of the year).
- At least one each of the Greek, Cyrillic and Hebrew text font families (could someone please point me to the 'best' font of those that are available?).
- The `manfnt` (requested by Phil Taylor).

Every font (presuming 256 characters) takes about one day to complete. This does not sound like too much time, but unfortunately I also have other work to do :-)

I am still open for requests, but you may have to wait a couple of months.

For further reading

- On the METAFONT language:* Donald E. Knuth, "The METAFONTBook". Addison-Wesley Publishing Company, June 1986, 361 pages.
- On using METAFONT to design real life fonts:* Donald E. Knuth, "Computer Modern Typefaces (Computers and Typesetting, volume E)". Addison-Wesley Publishing Company, June 1986, 588 pages.
- On the PostScript Language:* Adobe Systems Inc, "The PostScript Language Reference Manual". Addison-Wesley Publishing Company, December 1990, 764 pages.
- On METAPOST:* John Hobby, "A User's manual for METAPOST", AT&T Bell Laboratories Computing Science Technical Report 162, 1992. Comes as part of the METAPOST distribution.
- On Type 1 fonts:* Adobe Systems Inc, "Adobe Type 1 Font Format". Addison-Wesley Publishing Company, June 1995.
- On the Metafog program:* Richard J. Kinch, "Converting METAFONT Shapes to Outlines". Paper presented at the 1995 TUG Conference in St Petersburg, Florida, USA. Appeared in print in *TUGboat* 16.3

Threshing EPS files

Bogusław Jackowski, Piotr Pianowski, and Piotr Strzelczyk

BOP s.c.

ul. Piastowska 70, Gdańsk, Poland

B.Jackowski@gust.org.pl\ P.Pianowski@gust.org.pl\ P.Strzelczyk@gust.org.pl

Abstract

In this article we describe the CEP package for compressing EPS files. It belongs to the public domain and was released at the GUST meeting in Bachotek, 1997.

The amount of disk space occupied by bitmap graphics is a well-recognized problem. For example, a 300 dpi picture (A4) contains ca 8700000 pixels; assuming that each CMYK pixel occupies four bytes, one obtains ca 35MB of disk space needed to store the picture.

Now, imagine a T_EX-er, who is not allowed to use binary graphic data (because of the otherwise magnificent DVIPS); thus our poor T_EX-er usually converts the binary data to hexadecimal EPS files, thus doubling the required space, and next, after compiling a document with T_EX+DVIPS, the whole graphic data is put into the resulting PostScript file, so the required space is doubled again — altogether 140MB per one A4 page. The nightmare begins...

This problem is not a new one; it was recognised by Adobe a relatively long time ago. In the PostScript Level 2 specification, they included objects called filters which enable data compression. In particular, instead of hexadecimal data, one can use ASCII85 encoding (there are explanations of abbreviations at the end of the article), run length compression, LZW compression, DCT (used in JPEG files), and many others. Why not make use of these tools? The question is not as silly as it may look at the first glance, as there exist relatively few applications capable of generating well-compressed PostScript graphics.

We decided to patch somehow this gap. We developed a little package enabling the compression of “normal” (non-compressed) graphic data. The nature of the problem is more complex, however, than one might expect. In particular, a universal, always efficient compression technique does not exist. Choice of an optimal algorithm depends upon the kind of data, form of the file and the expected application. Hence, the package has several “buttons” which enable controlling various aspects of compression.

Actually, the name CEP is derived from “compressed EPS”. Coincidentally, the name in Polish

means “the flail”. We hope that others find threshing EPS files useful, in order to get rid of chaff, i.e., redundant data.

About the program

Our package consists of two pairs of AWK programs (`cep.awk-uncep.awk` and `cop.awk-uncop.awk`), four MS-DOS batch files and text information. `cep.awk` and `cop.awk` generate (on-the-fly) PostScript programs which, processed by Ghostscript, yield the appropriate data compression. UNCEP and UNCOP accomplish (using a similar technique) the reverse process, i.e., uncompression.

CEP is devised for the compression of the usual bitmapped EPS files, containing a single, hexadecimally-coded image; COP can be used to compress any PostScript data.

The question arises: Why use two packing techniques? The answer is simple: the efficiency of compression is higher if a compression program knows in advance which kinds of data are to be expected. In general, bitmaps are more regular (redundant) than arbitrary PostScript data, hence even simple algorithms turn out to be more efficient.

Tests show that in the best case (screen dumps) squeezing up to 10% of the original size is nothing unusual. Sometimes, however, no compression method gives a satisfactory result. In such a case, one can always use encoding data using the ASCII85 filter, obtaining a reduction of a hexadecimal bitmap size by approximately 35%.

Below we give a brief description of CEP and COP. So far, only the MS-DOS version of the PostScript-compressors is available, but it should be easy to adapt our package to any platform, where GAWK and Ghostscript are available. In this version the GNU implementation of AWK (`GAWK-EMX.EXE`) and Aladdin Ghostscript interpreter (`GS386.EXE`) are used.

We tested the package using several Ghostscript and GAWK implementations; now we use Ghostscript 5.10 and GAWK 3.0.3.

CEP

The CEP subpackage consists of the MS-DOS batch files `cep.bat` and `uncep.bat` and the AWK programs `cep.awk` and `uncep.awk`. First, AWK inspects the source EPS file doing its best to recognize a position of a hexadecimal bitmap; next it creates an appropriate PostScript program; and then the control is passed on to Ghostscript which just performs the submitted program: encodes the bitmap and copies verbatim the remaining lines. The original preamble is slightly modified; nevertheless, all DSC comments are left intact.

If the bitmap cannot be found or the AWK suspects that troubles may arise, the CEP engine gives up.

The resulting file should be verified prior to removing the original one, as the CEP heuristic tricks may fail to fix the bitmap properly; moreover, due to Ghostscript bugs, premature removal of the source may also be painful.

CEP never generates binary output — only hexadecimal or ASCII85 encoding are supported. This is due to the fact that CEP-compressed EPS files are primarily meant to be used by \TeX +DVIPS. Nevertheless, the resulting files can be used in other typesetting systems as so-called placeable EPS files. The applicability to non- \TeX applications, however, is somewhat limited, as binary TIFF previews (required by WYSIWYG applications) may be misinterpreted by (G)AWK.

UNCEP requires that a CEP-compressed file was not changed. In particular, it relies on the information in a quasi-DSC comment `%UNCEPInfo:.` This information can be destroyed by a seemingly innocent modification (e.g., by adding or removing a comment line). Note that the technique employed by CEP destroys, by its nature, the information about the line-breaking structure of the hexadecimal bitmap. Therefore, UNCEP cannot retrieve the original file. Line-breaking structure does not make any problem for a PostScript interpreter. There exist programs, however, that read their own bitmapped EPS files, which for unknown reasons make use of such (sub)lexical information; Aldus PhotoStyler is a notable example.

The command line invoking CEP is pretty simple:

```
cep.bat <in_file> <out_file> <options>
```

One should remember that the names of input and output files must differ. The program recognizes the following options:

- `s` — use ASCII85 coding (default)
- `h` or `H` — use HEX (hexadecimal) coding
- `r` or `R` — use RLE (RunLength) compression (default)
- `l` or `L` — use LZW compression
- `f` or `F` — use Flate compression (PDF and Level 3)
- `n` or `N` — don't compress

Invoking UNCEP is even simpler:

```
uncep.bat <in_file> <out_file>
```

As mentioned, decompression and decoding methods are taken from an input file.

COP

The subpackage consists of the MS-DOS batch files `cop.bat` and `uncop.bat`, and the AWK programs `cop.awk` and `uncop.awk`. COP reads and encodes appropriately the supplied data. No analysis of the PostScript data is performed, as the entire file is encoded without changing even a bit. The only aspect that is taken into account is the DSC comment `%%BoundingBox:;` if it is found, COP inserts this comment in the preamble, otherwise the resulting file does not contain the bounding box information.

COP-generated files are readable by any PostScript Level 2 interpreter.

UNCOP scans the header and deduces from it the method of decompression, hence no options are needed. UNCOP, unlike UNCEP, retrieves precisely the original file. It is still recommended, however, that a user verifies whether the resulting file is properly interpreted by Ghostscript. Due to Ghostscript bugs, premature removal of the source file after compression or decompression may turn out to be painful.

Since COP can be used to compress any data for arbitrary applications, binary encoding is allowed also. The resulting files can be used with typesetting systems that accept so-called placeable EPSs. Unfortunately, binary TIFF previewers make files after compression illegible for PostScript.

The usage of COP is similar to that of CEP:

```
cop.bat <in_file> <out_file> <options>
```

The program recognizes the following options:

- `s` — use ASCII85 coding (default)
- `b` or `B` — use binary coding
- `h` or `H` — use HEX (hexadecimal) coding
- `r` or `R` — use RLE (RunLength) compression (default)

- l or L — use LZW compression
- f or F — use Flate compression
(PDF and Level 3)
- n or N — don't compress

Observe that binary encoding is, in fact, no encoding at all.

The reverse process, i.e., UNCOP decompression, is also straightforward:

```
uncop.bat <in_file> <out_file>
```

As with UNCEP, decompression and decoding methods are taken from an input file.

A heap of remarks concerning our package

The applied solution addresses several problems:

1. It is not at all obvious how to determine syntactically where a hexadecimal bitmap begins in an EPS file; semantic analysis (by redefining PostScript primitives `image`, `imagemask` and `colorimage`) is possible, but it also has its limitations; anyway, we decided to recognize a bitmap syntactically, which implied a problem of recognizing such artifacts as `add` or `def` which look like fragments of a bitmap but, in fact, are not.
2. Also, it is not obvious which compression method should be applied for a given data type; usually, ASCII85 encoding is advisable; for pure bitmaps (CEP) RLE compression is satisfactory, although LZW and Flate filters usually produce much better results (the latter seems to be the best); nevertheless, both LZW and Flate encodings have limited usability:
 - (a) LZW encoding is not implemented in Ghostscript ver. > 4 due to USA patent law; as a by-pass, Aladdin implemented an LZW-compatible filter which produces non-compressed data (in fact, enlarged by some 10%) readable for any `LZWDecode` filter. You can use an old Ghostscript version, or compile a Ghostscript version containing the real LZW filter at your own risk, but...
 - (b) Flate encoding (the same that is used in GZIP) is available on photo-typesetters having implemented PostScript Level 3; it is also used in PDF files. It is safe to assume that Ghostscript ver. > 4 has this filter built-in. With other PostScript devices, in particular commercial ones, the test described in point 6 may prove useful.

As a rule of thumb we would suggest not to use any compression but ASCII85 for detailed

colour photo images. It is just a weakness of all non-lossy techniques — algorithms employed by ARJ, ZIP, LHARC, and others would yield poor results also. A reasonable alternative for data of this kind would be DCT (JPEG) compression.

3. As was mentioned above, ASCII85 encoding can usually be recommended; it added, however, some troubles. First, due to Ghostscript bugs, we decided to add the (dummy) `NullEncode` filter which seems to cure the problem. But there is one more problem: ASCII85-encoded bitmaps may contain lines looking like DSC comments, i.e., they may begin with double percent signs, `%%`, or with a percent-exclamation sign pair, `%! — why didn't Adobe exclude the percent from ASCII85? Some programs may try to interpret pseudo-DSC lines. For example, DVIPS just removes such lines, unless the option -K0 is not used; on the other hand, leaving DSC comments intact may stupefy document managers.`
4. It would be convenient to have some more filters implemented, in particular DCT and CCITT-Fax; both of them, however, make use of some additional input data which makes using them more complex; moreover, it is not clear whether one can find the optimal compression parameters for DCT without a WYSIWYG program; we consider a possibility of one-to-one conversion between JPEG files and EPS files making use of DCT filters; also, a similar conversion between GIF files and EPS files making use of LZW filters can perhaps be implemented.
5. The package conserves the working disk space — no large temporary files are created; roughly, the needed disk space is equal to the size of the source plus the size of the target.
6. The following file may be helpful for verifying whether a given PostScript device is able to interpret compressed EPS files:

```

%!PS-Adobe-2.0 EPSF-1.2
%%Pages: 1
%%BoundingBox: 0 0 540 150
%%EndComments
/Helvetica 8 selectfont
90 rotate
1 2 moveto
(*)
{0 -10 rmoveto gsave show grestore}
255 string
/Filter
resourceforall
showpage

```

`%EOF`

Running this program yields the list of filters for a given device. The error reported during the processing of this file proves that the device is not Level 2 compatible. In such a case, using the CEP package should be abandoned.

7. Bugs and traps:

- (a) Apparently, by preceding the `closefile` command by `flushfile`, one neutralizes an error in GS 3.x (tail of output swallowed).
- (b) Adding (a dummy) `NullEncode` filter neutralizes (probably) another Ghostscript bug: an `ASCII85Encode` filter with a target procedure may produce superfluous EOD marks, i.e., “~>” (if things go really badly you can obtain thousands of them). Using the target procedure instead of a file object excludes GS ver. < 3.x, because early Ghostscripts didn’t support all features of PostScript Level 2. Nevertheless, Ghostscript ver. ≥ 2.6 can be used for compression with hexadecimal encoding (it has a “legal” LZW compression).
- (c) The target procedure mentioned in 7b is, in turn, due to special treatment of the ASCII85-encoded lines that look like DSC comments; this special treatment is breaking lines after the first percent character. It is dedicated to the DVIPS driver which has a dangerous option `remove comments` (-K1).
- (d) An artificial form of quitting, i.e., `{2 2 .quit}` instead of `{2 .quit}`, is due to an infinite loop of Ghostscript 3.5x caused by the latter form. The Ghostscript internal operation `.quit` was chosen to provide error handling at the level of the operating system.
- (e) Still, there exist bugs in older Ghostscript that we were not able to neutralize; e.g., some EPS files are properly compressed by GS 2.6, but Ghostscript 2.6 breaks while displaying them; GS 3.51 behaves similarly with other bitmaps. So far, Ghostscript > 4 seems to be the most resistant to the “filter trial”, but it also reveals some deficiencies. Let’s hope that GS 5.5, which is expected to appear soon and is claimed to have most of PostScript Level 3 features implemented, will be still better.
- (f) Summing up, we would strongly recommend using Ghostscript 4.x or 5.x (pos-

sibly with `LZWEncode` compiled in) and GAWK 3.x: GS 4.x is nearly complete implementation of the Level 2 PostScript; GAWK 3.x provides regular expressions for record separators, which makes it possible to force handling end-of-lines in exactly the same manner as PostScript does and, moreover, is more reliable than earlier versions.

CEP for everybody

The packages described herein, we have developed for our own purposes. We make them available to the public in the hope that others will find them useful too. We intend to support the packages, but we cannot guarantee that we will be able to follow the frequency of Ghostscript upgrades.

Note that all copyrights, copylefts, copyups, copydowns, or whatever you wish to call them, concerning all the files in the CEP/COP packages are essentially of the public domain character.

Vocabulary

You may find useful the following short explanation of terms appearing throughout the article.

Ghostscript, GS: A reliable and efficient interpreter of PostScript language by Aladdin Enterprises, available as a free public license product; its current version (in July — 5.10) turns out to be much more reliable than not a few commercial interpreters.

AWK: A popular utility and a programming language for convenient and efficient batch data-reformatting; written in 1977 by Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan.

GAWK: GNU AWK, GNU Free Software Foundation implementation of AWK, written in 1986 by Paul Rubin and Jay Fenlason, with advice from Richard Stallman.

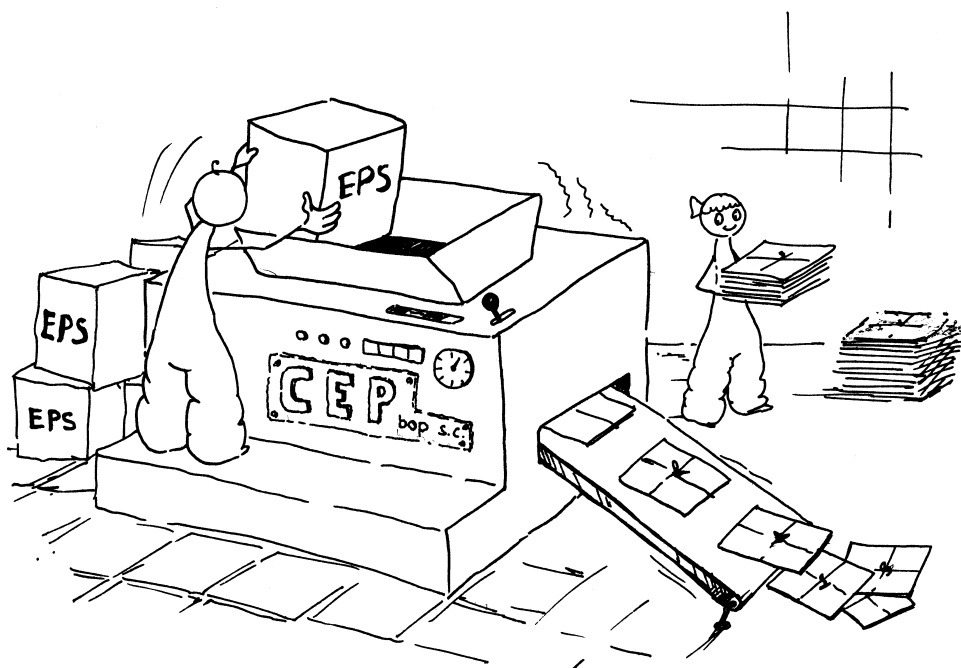
GNU: The initiative of the Free Software Foundation (FSF), a non-profit organization dedicated to the production and distribution of freely distributable software, founded by Richard M. Stallman.

T_EX: Public domain typesetting system by Donald E. Knuth of Stanford University.

DVIPS: A popular T_EX-to-PostScript driver by Tomas Rokicki of Stanford University.

DSC: Document Structuring Convention — a standard for structuring PostScript documents designed by Adobe.

- ASCII85:** Algorithm for coding binary data as 7-bit ASCII text consisting of only printable characters; encodes every four bytes as five characters from % to u; additionally z is used to code four zeros (see PostScript Language Reference Manual, second edition, pp. 128–130).
- RLE:** Run Length Encoding — a standard method of data compression (see PostScript Language Reference Manual, second edition, pp. 133–134).
- LZW:** An algorithm of data compression by J. Ziv, A. Lempel (1978), improved by T. Welch (1984); Unisys, at the time Welch’s employer, was granted a US patent in 1985 on Welch’s algorithm; a grandfather clause was established by Unisys to make pre-1995 implementations of LZW code free of royalty requirements, thereby eliminating such claims on UNIX compress (information from Nelson H. F. Beebe, e-mail: beebe@math.utah.edu).
- DCT:** Discrete cosine transform compression, an elaborate, very efficient but lossy compression scheme, used in JPEG file format.
- JPEG:** Joint Photographic Experts Group, an organization responsible for developing an international standard for compression of image data; the PostScript (Level 2) `DCTEncode` filter conforms to the JPEG-proposed standard.
- GZIP:** Compressing utility by GNU Free Software Foundation, based on a superior and unpatented compression algorithm (modified Lempel and Ziv algorithm), developed in order to get rid of the patented LZW algorithm. It has become a standard compression tool in UNIX systems.
- Flate:** Filter in PostScript Level 3 based on the compression algorithm used by GZIP; the name is the truncation of the words “inflate” and “deflate”.



More T_EX-PostScript links

Bogusław Jackowski, Piotr Pianowski, Piotr Strzelczyk

BOP s.c.

ul. Piastowska 70, Gdańsk, Poland

B.Jackowski@gust.org.pl, P.Pianowski@gust.org.pl, P.Strzelczyk@gust.org.pl

Introduction

According to Donald E. Knuth's decision, T_EX stays frozen. This does not mean, however, that it cannot be improved. There are several ways to conform to Knuth's idea of keeping T_EX frozen while improving it at the same time:

- developing macro packages;
- writing utility programs: drivers, pre- and post-processors of DVI files, programs for generating T_EX documents, graphic utilities (such as METAPOST), etc.;
- providing links to other languages and/or systems: RTF, PDF, HTML, SGML, PostScript, databases (bibliography), WWW pages, etc.

We shall focus our attention on one of the many aspects, namely on PostScript applications. Linking T_EX and PostScript was a giant step towards making a professional typesetting system out of T_EX. PostScript and T_EX fit excellently together, as PostScript is a powerful, well-defined, world-wide standard language of graphic and text page description, and so is T_EX. Since most phototypesetters and many printers understand PostScript, it is crucial that T_EX also understands PostScript.

Of course, the basic link is a good PostScript driver. Fortunately, such a driver exists — it is Tom Rokicki's `dvips`. But the driver alone is nowhere near enough — it provides access to nearly all of PostScript's features, but many of these need extra tools in order to make them easy to use. The more so as PostScript, unlike T_EX, continues to develop rapidly (recently, Adobe released PostScript Level 3) and thus one can presume that more and more new tools will be needed.

We describe here the tools we have developed for our own purposes. We make them available to the public in the hope that others will find them useful too. The tools were released at the GUST meeting in Bachotek, 1998. The release can be considered as a continuation of a series of earlier releases of similar tools, such as the `PS_VIEW` previewer, the `CEP` utility for compressing EPS files, `EPS2MF` and `MF2EPS` converters, and others.

Tiff2ps

Encapsulated PostScript files (EPS) are commonly used with T_EX for including graphics. Unfortunately, not all systems support encapsulated PostScript. It is understandable, since in order to interpret EPS files, a nearly complete PostScript interpreter is necessary. Therefore, some applications prefer simpler formats. Perhaps one of the most popular is TIFF: a tag-based file format for storing and interchanging raster images. Despite being simpler than EPS, TIFF is rich enough to describe a broad range of bitmap images.

In order to convert a TIFF file to an EPS file a special program is needed. There are several programs available, but we do not know of any written in PostScript. We decided to write our `tiff2ps` converter in PostScript (actually, in Ghostscript) for several reasons: (a) PostScript has fundamental compression algorithms implemented, which simplifies the processing of TIFF data; (b) the portability of PostScript programs is higher than that of those written in C, and comparable with the portability of programs written in T_EX; (c) by definition, PostScript programs exist only in source form and thus modifications and enhancements by third parties are possible; and last but not least, (d) employing Ghostscript guarantees surprisingly efficient processing.

The `tiff2ps` converter accepts most TIFF files conforming to the TIFF 6.0 specification, including gray, palletted, RGB, CMYK colour models, and LZW, RLE, CCITT (fax) compression; JPEG compression is expected to be available soon.

The resulting EPS files can be compressed using LZW, RLE, or Flate PostScript filters; moreover, the resulting bitmap can be written in either a hexadecimal or an ASCII85 encoded form.

The package also can be used for generating colour-separated EPS files (out of CMYK TIFFs) and "EPS thumbnails", i.e., EPS files with a reduced resolution. Moreover, EPS headers, containing only a pointer to a source TIFF file, can be created. Such an approach has many advantages, as headers are usually negligibly small, which increases the

efficiency of the processing of documents and saves disk space. This form is similar to the OPI (*Open Prepress Interface*) standard used in prepress systems. One should be aware, however, that not all PostScript devices (phototypesetters) are equipped to accept such header files.

The present version of the `tiff2ps` converter is under development — more facilities and more TIFF formats are to be implemented; nevertheless, backward compatibility will be preserved.

Pf2afm

A “canonical” PostScript Type 1 font comes in two files: an AFM (*Adobe Font Metrics*) file and a PFB or PFA file (*PostScript Font Binary* or *PostScript Font ASCII*, respectively). PFB and PFA files contain exactly the same information, namely the description of glyph shapes; the only difference being that PFB — as the name suggests — contains the data in a binary form, while PFA exploits ASCII (hexadecimal) representation of the data. The most important part of an AFM file contains information about the dimensions of glyphs and about kern pairs.

PostScript interpreters make no use of AFM files. The information is — as Adobe says — for “communicating font metric information to people and programs.”

\TeX takes metric information from TFM files, not from AFM ones. Fortunately, the AFM format is so general that it can serve not only for “application programs that generate PostScript language page descriptions”; it can also be used by auxiliary programs that prepare metric data for typesetting systems. One of such programs is the well-known `afm2tfm` (written by T. Rokicki and D. E. Knuth) which produces TFM files out of AFM ones and thus makes PostScript fonts available for \TeX users.

It should be emphasized that for \TeX , users having both PFB/PFA *and* AFM files is crucial. Alas, with the advent of Windows systems, a new form of font metrics emerged, namely PFM (*Printer Font Metrics*), containing a subset of the information stored in AFM files. PFM files are used by Adobe Type Manager (ATM) for Windows.

As a result, some vendors started to distribute PostScript Type 1 fonts with PFM files instead of AFM ones. We looked for a reliable program to convert PFM files to TFM ones, but we found none. Although a few programs converting PFM to AFM exist, we were not satisfied with them. We were thinking about writing our own converter, when we encountered in the Ghostscript distribution a PostScript program `printafm`, written at some

point by James Clark. The program is capable of extracting metric information from a PostScript font and writing the data in an AFM form. We decided to enhance Clark’s program in such a way that it could make use of the data stored in a related PFM file, whenever available. Moreover, we added an interface facilitating batch processing.

The result of the enhancement is the `pf2afm` converter, or, more adequately, the `pf2afm` patch. Figure 1 shows the place of `pf2afm` in a simplified \TeX file processing scheme.

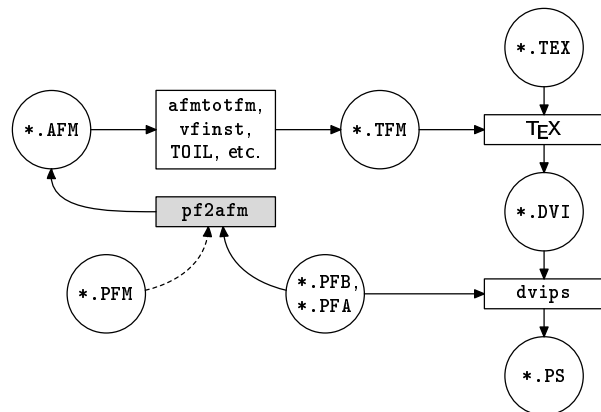


Figure 1: \TeX file processing scheme

In general, the retrieval of the complete metric information from PFB/PFA and PFM files is impossible; only if we are lucky, i.e., if the *Encoding* vector of a given font contains all the characters we need, can we conveniently use the resulting AFM; otherwise, hand-tuning may turn out to be necessary.

We believe that such a patch (or converter) may prove useful for people dealing with PostScript fonts, not only for \TeX users. Anyway, the `pf2afm` tool has been included in the standard Ghostscript distribution.

In our practice, on several occasions, we encountered situations when missing AFM files caused trouble. Before we created the `pf2afm` converter, we had had to use special font programs, such as Fontographer, which we would gladly avoid, to a large extent because of the low reliability of much “professional” software.

Ttf2pf

When a TrueType font format (TTF) was introduced by Apple and Microsoft, Adobe responded by equipping PostScript with a TrueType substitute, Type 42 font format, and by including a TrueType rendering engine in their PostScript interpreter. The relationship between TrueType and Type 42 is schematically shown in Figure 2.

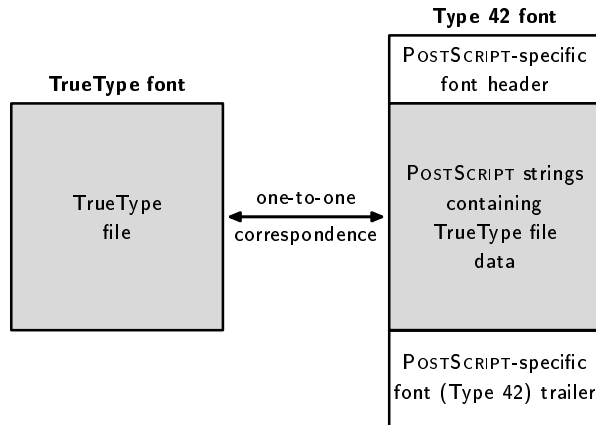


Figure 2: The relationship between TrueType and Type 42 formats

The details are unimportant here. We only observe that Type 42 is, in fact, an equivalent of the TrueType format: TrueType data are simply stored inside a Type 42 file, i.e., no surgery on font intestines takes place and the intact TrueType data can be retrieved from the Type 42 file. Note that the conversion between Type 1 and Type 42/TrueType involves fundamental changes of the representation of glyphs, in particular the hinting information cannot be preserved.

In other words, the conversion between TrueType and Type 42 is purely formal, something like the conversion between PFB and PFA, although the latter is significantly simpler.

Since the font market has been recently flooded with TrueType fonts, we decided that they should be available for \TeX users. The best starting point — as usual — turned out to be Ghostscript. Actually, the `ttf2pf` converter is based on PostScript programs to be found in a standard Ghostscript distribution; `ttf2pf` itself will be probably included in the standard Ghostscript distribution.

The `ttf2pf` converter generates two files: Type 42 and AFM; the Type 42 file appears in an ASCII form, thus it can be included by `dvips` as an ordinary header file; the AFM file allows the generation of a TFM file using standard tools.

It should be noted that not all phototypesetters cope with Type 42 fonts. We also had trouble in converting PostScript files containing Type 42 fonts to PDF format using Adobe's Acrobat Distiller. A careful inspection showed that they were properly embedded in PDF files as genuine TrueType fonts, but the Acrobat Reader displayed uniform rectangles instead of glyphs (although it was able to rec-

ognize that a document contained TrueType fonts). That's it for the bad news.

For the good news: Ghostscript renders Type 42 fonts smoothly. Incidentally, there is a possibility that during the TUG meeting in Toruń the Ghostscript release compatible with PostScript Level 3 will be available.¹

Colormap

Occasionally, a modification of a bitmap graphic is necessary. For example, one may wish to have a pale version of a scanned photo in order to use it as a background. Such an intervention can be easily accomplished using special graphics programs, but this means that both the original and the modified images need to be stored, which is inconvenient. Moreover, GUI programs usually do not allow users to define such changes numerically.

Fortunately, modifications of this kind can be performed by a PostScript engine, and thus it is possible to also perform them at the \TeX level; `colormap` is a tiny package of \TeX macros that makes possible nearly arbitrary colouring of gray bitmap images.

The basic method of specifying colour change is to define the colours to which black and white should be mapped, assuming that for the intermediate colours a linear interpolation is applied. You can specify the mapping using either gray or CMYK

¹ Ghostscript 5.50 (released a few weeks after the conference) is still not fully compatible with PostScript Level 3 but has a lot of its features.

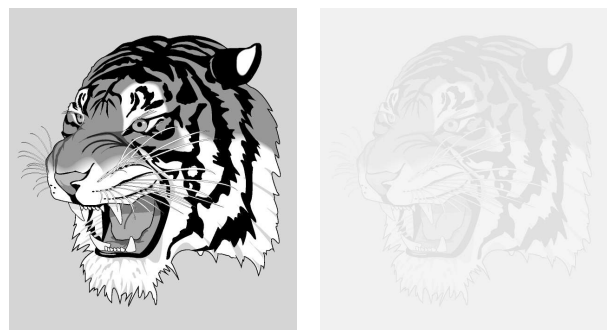


Figure 3: The picture to the left is an original image, placed using the command `\epsffile{tiger.eps}` (assuming the usage of the `epsf` package from the standard `dvips` distribution); the picture to the right is obtained using the command `\lingraymap[.85:.95]{\epsffile{tiger.eps}}`, where `\lingraymap` is defined in `colormap`.

models. For example, in order to make an image pale, one should map black to ca 15% of black and white to ca 5% of black, i.e., following the PostScript custom, to 85% and 95% of gray, respectively. Figure 3 shows the result of such a modification.

The linear interpolation of colour is not obligatory. It is possible to apply an arbitrary mapping, given by an array (having 256 entries), represented as a PostScript hexadecimal string. The string can be either created manually or computed by an auxiliary program.

Actually, the `colormap` package defines four macros, allowing users to map a grey-scale image to:

- another grey-scale image using a linear interpolation (`\lingraymap`)
- a CMYK-model image using a linear interpolation (`\lincmykmap`)
- another grey-scale image using an arbitrary mapping (`\gengraymap`)
- a CMYK-model image using using an arbitrary mapping (`\gencmykmap`)

For details, see the file `colormap.tex`. Although the `colormap` macro package is written in plain TeX, it is also supposed to work with L^ATeX.

Acknowledgements

It should be emphasized that all the tools described in this paper could be developed only thanks to L. Peter Deutsch's marvelous interpreter of the PostScript language: it is Ghostscript that provided a convenient platform for creating such tools. We are grateful to L. Peter Deutsch for making Ghostscript available as freeware, for maintaining and developing it, and for helping us promptly whenever we met difficulties.

Postscript

Any trademarks, trade names, service marks, or service names owned or registered by any other company and used in this publication are the property of their respective companies.

METAPOST and patterns

Piotr Bolek

ul. Szkolna 15, 05-180 Pomiechówek, Poland

Phone: (48) 22-785 43 39

P.Bolek@ia.pw.edu.pl

Abstract

In this paper the METAPOST macros for defining and using patterns are presented. METAPOST is an excellent graphics program which gives the user access to many PostScript features. But there is no way to access the *Pattern Color Space* of PostScript Level 2. The `mpattern` package is the author's attempt to give users of METAPOST a comfortable way of accessing this feature of PostScript. This package allows the user to define patterns using arbitrary METAPOST code, modify the pattern transformation matrix and specify vertical and horizontal displacement of adjacent pattern cells. Examples of defining and using patterns are shown.

Introduction

METAPOST is a very good graphics program. It takes the best from METAFONT, PostScript and T_EX. From METAFONT, it borrows the declarative programming model and a way of describing graphic objects. It has a very comfortable interface to T_EX and PostScript features. The user can typeset labels using T_EX commands and fonts and modify the graphic state parameters of PostScript, such as painting color, line thickness and dashing patterns, as well as the way of line ending and joining. But, the possibility of defining and using patterns is lacking.

Patterns are very useful and comfortable. Once defined, they behave like ordinary colors — they are automatically tiled and clipped on the edges of the painted area by the PostScript interpreter. They can be used for easy definition of textures such as stripes, waves, checkers, hexagons and many more.

Direct implementation of the interface to patterns in METAPOST is impossible without modifying the sources of the METAPOST program itself. The solution proposed by the author is different. The main part of the pattern package is written as METAPOST macros, but the figures in which the patterns are used must be postprocessed by a simple perl script. This script is a simple wrapper that calls the METAPOST program, finds the figures in which the patterns were used and postprocesses these figures. The user who wants to define and use patterns in METAPOST figures must use this wrapper script (called `mpp`, which stands for “METAPOST with Pat-

terns”) instead of direct invocation of the METAPOST program.

Patterns in PostScript

There are two types of patterns in PostScript — uncolored and colored. Uncolored patterns do not specify any color and act as stencils for painting with separately specified colors. In uncolored patterns, operators that specify colors are not allowed. The colored pattern specifies the colors used for painting the pattern cell.

Definition of patterns in PostScript consists of several elements. The pattern is defined as a special kind of dictionary. (The PostScript data structure acts as an associative array with elements which may have different types.) The main element of the pattern dictionary is `PaintProc` — the arbitrary PostScript procedure which is executed to paint a single pattern cell.

The other important elements of pattern definition are: pattern bounding box (`BBox`) which is used to clip the drawing made by `PaintProc` and horizontal and vertical spacing between adjacent pattern cells (`XStep`, `YStep`). This spacing may differ from the values implied by the dimensions of the pattern bounding box — the contents of adjacent cells will then overlap or there will be gaps between cells. Values of these parameters must be different from zero — either positive or negative.

The pattern shape can be modified by the arbitrary affine transformation specified during definition of the pattern — pattern cells can be scaled, rotated or skewed.

PostScript patterns and METAPOST

How is METAPOST related to PostScript pattern color space? The main part of the pattern definition is the `PaintProc` procedure. It is an arbitrary PostScript procedure—and the purpose of METAPOST is to produce arbitrary PostScript procedures. Therefore, the picture produced by METAPOST can be used as a definition of the pattern `PaintProc`. METAPOST knows the picture bounding box so we can also use this information if we need it.

METAPOST also can be used to specify the pattern transformation matrix which will be used to change orientation, size or shape of the basic pattern cell. The METAPOST `transform` type contains the same information as the PostScript transformation matrix. We can specify the transformation of a pattern cell using comfortable METAPOST (METAFONT) transformation expressions.

The `mpattern` package

The `mpattern` package is the interface to the PostScript Pattern Color Space from METAPOST. Using this package, we can define patterns using arbitrary METAPOST commands. We can also specify the bounding box of a pattern and spacing information (`XStep`, `YStep`). It is possible to use expressions of the type `transform` to specify an arbitrary affine transformation which will be applied to our pattern. The patterns defined with this package are colored patterns.

Once defined, patterns can be used in natural way—by using the `withpattern` operator, similar to `withcolor`, `withpen`, etc.

Here is a simple example. Assuming that we have already defined the path `bean`, we can define and use a pattern like that below:

```
beginpattern(checker);
  fill unitsquare scaled 4mm rotated 45;
endpattern;
beginfig(1);
  fill bean withpattern checker;
  draw bean;
endfig;
```

The result is shown on figure 1.

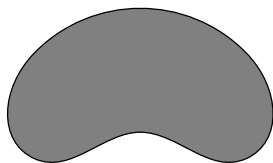


Figure 1

As we can see, the pattern is defined with two macros: `beginpattern` and `endpattern`. The former has one parameter—the name of a pattern. This name will be used later to identify the pattern when the user wants to use it as a filling for a closed path. Between these two macros, the user is allowed to use any valid METAPOST commands.

In our example, the pattern bounding box is not specified. In such situations, it is calculated by METAPOST, and in fact is identical with the bounding box of the implicitly defined picture. When the bounding box should be different from the default, we can specify it using the `patternbbox` macro. We can modify our pattern very easily by specifying the center of our square as the lower-left vertex of the bounding box. It will clip the basic cell of our pattern, ignoring everything outside the bounding box.

```
beginpattern(checker_clip);
  fill unitsquare scaled 4mm rotated 45;
  z1=llcorner currentpicture;
  z2=urcorner currentpicture;
  z1'=.5[z1,z2];
  patternbbox(z1',z2);
endpattern;
```

We can also change the spacing of the pattern without modification of its bounding box. We can specify the vertical and horizontal spacing separately (`patternxstep` and `patternystep` macros), or both of them at once (`patternstep`).

```
beginpattern(checker_ov1);
  fill unitsquare scaled 4mm rotated 45;
  patternxstep(4mm);
endpattern;
beginpattern(checker_gap);
  fill unitsquare scaled 4mm rotated 45;
  patternstep(6mm,7mm);
endpattern;
```

These three modifications of our first pattern are shown on figure 2.

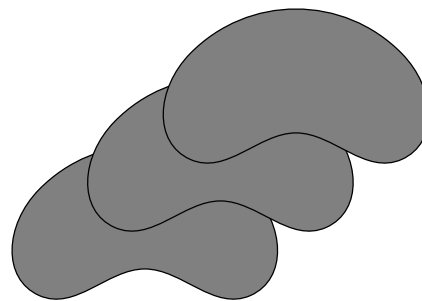


Figure 2

Using the `mpattern` package we can also specify arbitrary transformation of pattern cells. Ordinary METAPOST expressions of the type `transform` are used for this purpose. The `patterntransform` macro the user allows to specify the transformation of the pattern. The argument of this macro should contain the type `transform`. To rotate our example pattern we can define it as follows:

```
beginpattern(checker_rot);
  fill unitsquare scaled 4mm rotated 45;
  patterntransform(identity rotated 22);
endpattern;
```

Patterns may also be translated, scaled and slanted. The transformations can be joined in the usual way:

```
beginpattern(checker_sl);
  fill unitsquare scaled 4mm rotated 45;
  patterntransform(identity rotated 45
                  slanted .2);
endpattern;
```

Examples of transformed patterns are shown on figure 3.

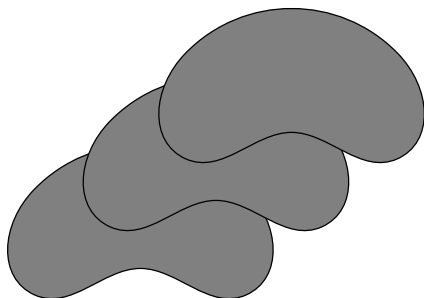


Figure 3

Now we know all the macros for defining and using patterns available to users:

- `beginpattern`, `endpattern`: the couple of macros which enclose the definition of basic pattern cell.
- `patternbbox`: the macro which allows the explicit specification of the pattern bounding box. If this macro is not used in the definition of the pattern, then the bounding box of the whole picture acting as the pattern cell is used. This macro may have two parameters of the type `pair`, or four numeric parameters.
- `patternxstep`, `patternystep`, `patternstep`: the macros for specifying spacing of the pattern cells.
- `patterntransform`: the macro for changing the shape of a pattern cell. The argument of this macros must have the type `transform` and it

represents the transformation which will be applied to the pattern. This transformation will take place *after* determining the bounding box and spacing of the pattern. Therefore the real size and shape of the basic pattern cell can be different from the ones specified in the pattern definition. Any valid METAPOST transform expression can be used as the argument of this macro.

- `patterncolor`: the macro used to assign color to the defined pattern. In the first stage of processing, the pattern is defined as a color which is replaced by the pattern itself during the postprocessing stage. The colors assigned to patterns are generated automatically, but we can force the use of concrete colors for this purpose. The argument of the `patterncolor` macro must be a number from range $[0, 1]$ and is interpreted as gray level (0 — black, 1 — white). Manual specification of a color tied with patterns requires that this color not be used for other (ordinary) purposes — because every object painted with this color will be painted with the pattern.

The use of only gray levels in the `patterncolor` macro is by the implementation of assignment of colors to patterns. The assignment information is stored in an array indexed by the colors, which is possible only when the colors are monochrome. This limitation may be relaxed in the future.

- `withpattern`: the primary operator which can be used for drawing shapes filled with earlier defined patterns. It can be used in a way similar to `withpen` or `withcolor` operators.

Implementation of the package

The `mpattern` package consists of two parts. The first is the METAPOST code in which the user interface and working macros are defined. The second is the simple perl script which invokes METAPOST and postprocess its output.

The processing of the patterns takes place in two steps and is managed by the simple perl script called `mpp`. In the first step, the METAPOST program is invoked. METAPOST code placed between `beginpattern`, `endpattern` macros is processed as the figure with a high number — the default is 999, but if this number is used by the user, then 998 is used, and so on. Problems can arise when the user uses all the picture numbers from 0 to 999, but hopefully this is a highly improbable situation. In the `endpattern` macro, the PostScript code generated by the pattern picture is read and remembered as

the PostScript pattern definition in the string variable. When the user uses the `withpattern` operator, this code will be placed at the beginning of the picture in which the pattern is to be used. This is performed with a `special` command. Information about every defined and used pattern is stored as comments in output files and in the log file. This information is used in the second step of processing patterns.

The PostScript code defining the pattern is constructed by the macro `endpattern`. The tiling information supplied by the user in the `patternbbox` and `pattern[xy]?step`¹ macros is converted to a form suitable for PostScript. PostScript code generated from an implicitly defined picture is used as the body of the pattern `PaintProc` procedure. All of these elements are placed together into a pattern dictionary and stored in string variables. The METAPOST transform expression, given as an argument to `patterntransform`, is converted to a PostScript transformation matrix and used in definition of the pattern to modify the coordinate system in which the pattern will be painted.

Every defined pattern is joined with the color which will be replaced by the pattern in the second step. The area to be filled with the pattern is filled by METAPOST with this color. The colors used for this purpose should not collide with “ordinary” colors used by the user. At the moment, colors which are to be replaced by patterns are constructed as

```
k * epsilon * white
```

where `k` is the number of the defined pattern, `epsilon` is the smallest number in METAPOST, and `white` is the white color. Using the macro `patterncolor` in the definition of the pattern, the user can explicitly specify the color (gray level) which will be used with the defined pattern. Ensuring that pattern colors will not collide with ordinary colors is, in this case, left to the user.

Every pattern is remembered in a variable with the same name as the argument of `beginpattern`, so the user should not try to use such a variable for other purposes.

The second step in processing patterns is performed by a perl script. Pictures in which patterns are used are found with the help of information stored in the log file. Then for every such picture, substitution from colors-to-patterns is performed. See the small example below.

If the pattern `checker` was defined first — “his” color will be `(epsilon * white)`, and the line `0.00002 setgray`

¹ Regular expression notation is used here.

in the output file will be replaced with `checker setpattern`

and definition of the pattern `checker` will be placed at the top of this file. If there are several different patterns used in one picture, then several pattern definitions will be placed at the top of output file.

Pattern examples

We have already seen several examples of patterns. These were checkers—the simplest possible ones. Now let us try to define more interesting and useful patterns.

Line patterns. The basic patterns are of course lines. It seems quite easy to define patterns from lines. But let us have a look at figure 4.

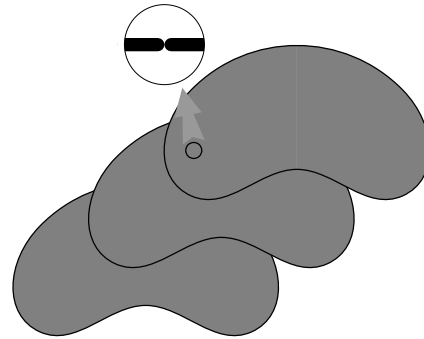


Figure 4

Patterns consisting of vertical and horizontal lines demonstrate something strange — at regular intervals they have thinner parts. This effect has two reasons — the pen used to draw lines is circular, and METAPOST calculates the bounding box of the pattern cell automatically. METAPOST is quite accurate when it calculates the bounding box of the picture, so we have what we wanted...

Knowing the reason for this unwanted effect, we can deal with it. Two possible solutions of this problem are shown in the listing below. The first solution is not to use a rounded pen (`lines_s`), and the second is to explicitly define the bounding box of the pattern in such a way that the rounded ends of the lines are cut off (`lines_ss`).

```
beginpattern(lines_h);
  draw origin--10left
    withpen pencircle scaled 2;
  patternystep(2mm);
endpattern;
beginpattern(lines_v);
  draw origin--10up
    withpen pencircle scaled 2;
  patternxstep(2mm);
```

```

endpattern;
beginpattern(lines_s);
  draw origin--10up
    withpen pensquare scaled 2;
  patternxstep(2mm);
  patterntransform(identity rotated -45);
endpattern;
beginpattern(lines_ss);
  draw origin--10up
    withpen pencircle scaled 2;
  patternxstep(2mm);
  patternbbox(left,10up+right);
  patterntransform(identity rotated 45);
endpattern;

```

Other patterns. Of course we are not limited to the definition of such simple patterns only. The following represent examples of two patterns and their transformations.

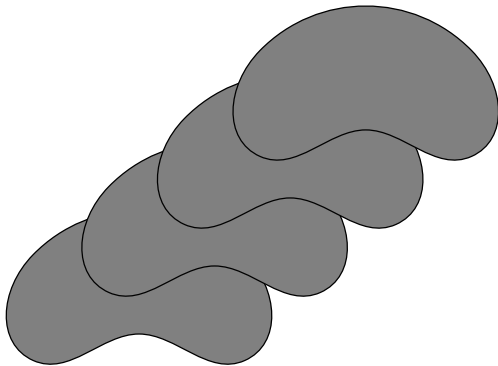


Figure 5

Definitions of our patterns are really simple, but they look quite interesting (fig. 5).

```

def wave_def=
  z1=origin; z2=5up+5right; z3=10right;
  draw z1{right}..z2..{right}z3;
  patternbbox(.25down,10right+5.25up);
enddef;
beginpattern(wave_i);
  wave_def;
endpattern;
beginpattern(wave_ii);
  wave_def;
  patterntransform(identity slanted .9
    rotated 35 xscaled 1.5);
endpattern;
def fish_def=
  z1=origin; z2=5right+5.up;
  path p; p=z1{up}..z2;
  draw p;
  draw p xscaled -1 shifted (5right+5up);
  draw currentpicture xscaled -1;

```

```

enddef;
beginpattern(fish_i);
  fish_def;
endpattern;
beginpattern(fish_ii);
  fish_def;
  patterntransform(identity slanted .9
    rotated 67 xscaled 1.5);
endpattern;

```

Interesting patterns can be defined using a for loop. Below we define a path which is later rotated, and as a result, we obtain quite interesting looking textures.

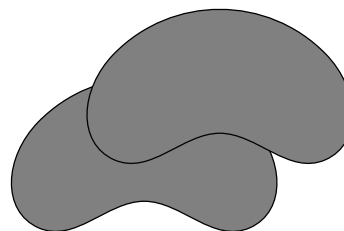


Figure 6

```

beginpattern(p_a);
  z1=(0,0); z2=5up;
  path p;
  p=z1{right rotated -10}..{up}z2;
  for i=1 upto 4:
    draw p rotated (i*360/4);
  endfor;
  patternbbox(-y2,-y2,y2,y2);
endpattern;
beginpattern(p_b);
  z1=(0,0); z2=5up;
  path p;
  p=z1{left}..z1+(-2,3)..{dir 65}z2;
  for i=1 upto 4:
    draw p rotated (i*360/4);
  endfor;
  patternbbox(-y2,-y2,y2,y2);
endpattern;

```

Colored patterns. All previously defined patterns were black and white; but we can of course define patterns with more colors. Here are three examples (fig. 7). (Editor's note: Figures 7 and 8 may be viewed in color at <http://www.tug.org/TUGboat/Articles/tb60/bolek-cfigs.pdf>.) Because the definitions of these patterns are not as simple as those above, it may be interesting to see the shape of the basic pattern cells (fig. 8). The code defining these patterns is contained in Appendix A.

Of course we can use fonts in our patterns. Quite interesting examples of patterns in which texts was used are shown in fig. 9.

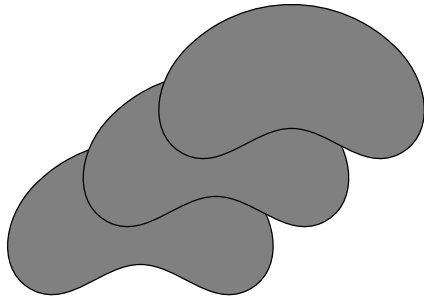


Figure 7

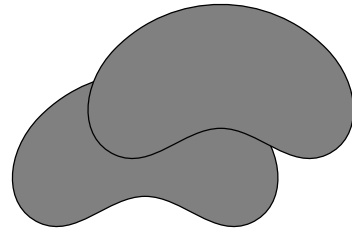


Figure 9

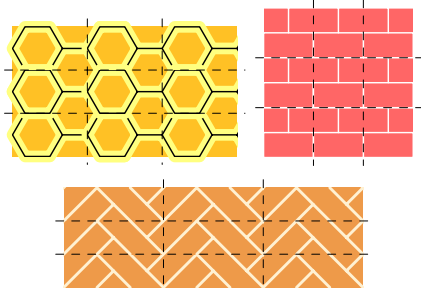


Figure 8

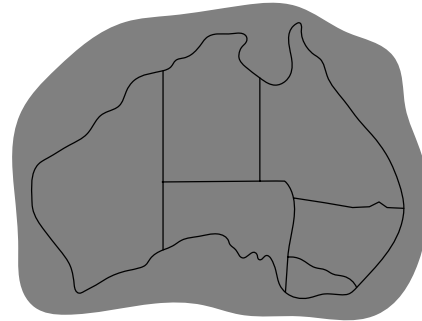


Figure 10

But using text in patterns may be dangerous. The PostScript created by processing the \TeX document including a figure with “text” patterns can cause errors in PostScript devices or interpreters. The problems occurs only when the fonts used in the patterns are bitmapped fonts. When we use Type 1 fonts, the file is processed without errors.

It seems that the dvips interface to PK fonts is not safe enough when used in patterns. So if we are going to use text in patterns, we should use only Type 1 fonts. Times New Roman is used in fig. 9.

The final example (fig. 10) is an illustration from a chapter about map coloring in book about combinatorics.

A Appendix

The definitions of patterns used in figure 7 are shown below.

```
beginpattern(Honey);
  path p;
  alpha:=360/6;
  for i=0 upto 5:
    z[i]=(3mm*up) rotated (i*alpha-30);
  endfor;
  p=z0--z1--z2--z3--z4--z5--cycle;
  z6=z5+z0-z1;
  z1b=(x2,y3); z2b=(x6,y0);
  patternbbox(z1b,z2b);
  fill z1b--(x1b,y2b)--z2b--(x2b,y1b)--cycle withcolor ((255, 193, 37)/255);
  drawoptions(withpen pencircle scaled 4 withcolor (red+green+.5blue));
  draw p;
  draw z5--z6;
  draw (z1--z2--z3) shifted (z6-z2);
  drawoptions();
  draw p;
  draw z5--z6;
  draw (z1--z2--z3) shifted (z6-z2);
endpattern;
```

```
beginpattern(Brick);
  u:=3mm;
  fill unitsquare scaled 2u withcolor (red+.4green+.4blue);
  draw unitsquare xscaled 2u yscaled u withcolor white;
  draw (u,u)--(u,2u) withcolor white;
  draw (0,2u)--(2u,2u) withcolor white;
  patternbbox(origin,(2u,2u));
endpattern;
```

```
beginpattern(Floor);
  u:=2mm;
  fill unitsquare xscaled 6u yscaled 2u withcolor ((238, 154, 73)/255);
  drawoptions(withpen pencircle scaled 1 withcolor ((255, 241, 210)/255));
  draw origin--(2u,2u)--(4u,0);
  draw (4u,2u)--(6u,0);
  draw (2u,0)--(3u,u);
  draw (5u,u)--(6u,2u);
  draw (-u,u)--(u,3u);
  draw (5u,3u)--(7u,u);
  patternbbox(origin,(6u,2u));
endpattern;
```

Patterns with texts from figure 9

```
beginpattern(txt_i);
  picture l;
  l=thelabel(btex\font\q=ptmr8r\q MetaPost etex, origin);
  draw l;
  z1=llcorner currentpicture;
  z2=urcorner currentpicture;
  draw l shifted ((y2-y1)*up+.5(x2-x1)*right);
  draw l shifted ((y2-y1)*up+.5(x2-x1)*left);
```

```
    patternbbox(z1,(x2,2[y1,y2]));
endpattern;

beginpattern(txt_ii);
  picture l;
  l=thelabel(btex\font\q=ptmri8r\q MetaPost etex, origin);
  draw l;
  z1=llcorner currentpicture;
  z2=urcorner currentpicture;
  draw l shifted ((y2-y1)*up+.2(x2-x1)*right);
  draw l shifted ((y2-y1)*up+.8(x2-x1)*left);
  patternbbox(z1,(x2,2[y1,y2]));
  patterntransform(identity rotated 60);
endpattern;
```

Improving T_EX's Typeset Layout

Hàn Thê Thành
Faculty of Informatics
Masaryk University
Brno, Czech Republic

Abstract

This paper describes an attempt to improve T_EX's typeset layout in P_dfT_EX, based on the adjustment of interword spacing after the paragraphs have been broken into lines. Instead of changing only the interword spacing in order to justify text lines, we also slightly expand the fonts on the line as well in order to minimise excessive stretching of the interword spaces. This font expansion is implemented using horizontal scaling in PDF. When such expansion is used conservatively, and by employing appropriate settings for T_EX's line-breaking and spacing parameters, this method can improve the appearance of T_EX's typeset layout.

Motivation

There exist many techniques which can be used to produce high quality typeset layout. Most of these are already implemented in T_EX, such as ligatures, kerning, automatic hyphenation, and very importantly the algorithm for breaking paragraphs into lines in an optimal way, generally known as “optimum fit”.

However, it is still a very difficult task to obtain a uniform level of grayness of the typeset layout, even with the help of these techniques. The primary reason is that it is not possible to ensure that all the interword spaces in different lines are the same. The “optimum fit” algorithm can break the paragraph into lines in the best way, but the amount of interword space depends strongly on many other parameters, such as the paragraph width, the tolerance of glue stretching/shrinking, the amount of interword glue, etc. Considerable effort is often required in order to adjust these parameters to achieve the appropriate break points and to reduce the contrast between the interword spaces in lines. The purpose of our experiment is an attempt to perform this task better by stretching or shrinking the fonts used in each line within reasonable limits. The idea is not really new, as it represents a quite common technique using electronic font scaling in order to expand text lines that do not fit the paragraph width. However this technique is also often regarded as a bad thing, since it is frequently (ab)used in order to rescue “impossible” cases, which often leads to overdoing the scaling and produces really ugly results. In our approach, we try to use this technique

in a rather different way: instead of using font scaling to improve only some “really bad” lines, we try instead to produce a “relatively good” paragraph, which does not contain any lines where the interword spacing is too bad. Then we apply font scaling to each line to reduce the difference between the interword spaces in lines. The limit of font scaling must, of course, be strictly controlled: in fact, the sum of the spaces between the words on a line is often very small in comparison to the sum of the character widths on the same line, so very slightly expanding the fonts may help considerably in improving the interword spacing.

This idea can easily be integrated with T_EX because of the biggest strength of T_EX – the “optimum fit” algorithm which is implemented in a very flexible manner, in order to handle restrictions on many various parameters in an optimal way. In particular, we perform the implementation in P_dfT_EX, where the font expansion is currently carried out by horizontal scaling in PDF as a first attempt. Other approaches may be attempted in the future as time allows.

Implementation

P_dfT_EX is based on the original source of T_EX, and employs the changefile mechanism which allows easy access to T_EX's internal data structures and simple modification of the relevant program code. Generating PDF output directly from T_EX is also an advantage for our task, as we can control the spacing much better than would have been the case had we attempted it via DVI. The process of adjusting interword spacing is as follows:

It was terribly cold and nearly dark on the last evening of the old year, and the snow was falling fast. In the cold and the darkness, a poor little girl, with bare head and naked feet, roamed through the streets. It is true she had on a pair of slippers when she left home, but they were not of much use. They were very large, so large, indeed, that they had belonged to her mother, and the poor little creature had lost them in running across the street to avoid two carriages that were rolling along at a terrible rate. One of the slippers she could not find, and a boy seized upon the other and ran away with it, saying that he could use it as a cradle, when he had children of his own. So the little girl went on with her little naked feet, which were quite red and blue with the cold. In an old apron she carried a number of matches, and had a bundle of them in her hands. No one had bought anything of her the whole day, nor had any one given her even a penny. Shivering with cold and hunger, she crept along; poor little child, she looked the picture of misery. The snowflakes fell on her long, fair hair, which hung in curls on her shoulders, but she regarded them not.

It was terribly cold and nearly dark on the last evening of the old year, and the snow was falling fast. In the cold and the darkness, a poor little girl, with bare head and naked feet, roamed through the streets. It is true she had on a pair of slippers when she left home, but they were not of much use. They were very large, so large, indeed, that they had belonged to her mother, and the poor little creature had lost them in running across the street to avoid two carriages that were rolling along at a terrible rate. One of the slippers she could not find, and a boy seized upon the other and ran away with it, saying that he could use it as a cradle, when he had children of his own. So the little girl went on with her little naked feet, which were quite red and blue with the cold. In an old apron she carried a number of matches, and had a bundle of them in her hands. No one had bought anything of her the whole day, nor had any one given her even a penny. Shivering with cold and hunger, she crept along; poor little child, she looked the picture of misery. The snowflakes fell on her long, fair hair, which hung in curls on her shoulders, but she regarded them not.

Figure 1: Parameters used in this experiment: `\pdfadjustlimit = 50`, `\tolerance = 200`, `\spaceskip = \fontdimen2\font plus 2\fontdimen3\font`

It was terribly cold and nearly dark on the last evening of the old year, and the snow was falling fast. In the cold and the darkness, a poor little girl, with bare head and naked feet, roamed through the streets. It is true she had on a pair of slippers when she left home, but they were not of much use. They were very large, so large, indeed, that they had belonged to her mother, and the poor little creature had lost them in running across the street to avoid two carriages that were rolling along at a terrible rate. One of the slippers she could not find, and a boy seized upon the other and ran away with it, saying that he could use it as a cradle, when he had children of his own. So the little girl went on with her little naked feet, which were quite red and blue with the cold. In an old apron she carried a number of matches, and had a bundle of them in her hands. No one had bought anything of her the whole day, nor had any one given her even a penny. Shivering with cold and hunger, she crept along; poor little child, she looked the picture of misery. The snowflakes fell on her long, fair hair, which hung in curls on her shoulders, but she regarded them not.

It was terribly cold and nearly dark on the last evening of the old year, and the snow was falling fast. In the cold and the darkness, a poor little girl, with bare head and naked feet, roamed through the streets. It is true she had on a pair of slippers when she left home, but they were not of much use. They were very large, so large, indeed, that they had belonged to her mother, and the poor little creature had lost them in running across the street to avoid two carriages that were rolling along at a terrible rate. One of the slippers she could not find, and a boy seized upon the other and ran away with it, saying that he could use it as a cradle, when he had children of his own. So the little girl went on with her little naked feet, which were quite red and blue with the cold. In an old apron she carried a number of matches, and had a bundle of them in her hands. No one had bought anything of her the whole day, nor had any one given her even a penny. Shivering with cold and hunger, she crept along; poor little child, she looked the picture of misery. The snowflakes fell on her long, fair hair, which hung in curls on her shoulders, but she regarded them not.

Figure 2: Parameters used in this experiment: `\pdfadjustlimit = 50, \tolerance = 500, \spaceskip = \fontdimen2\font plus\fontdimen3\font`

It was terribly cold and nearly dark on the last evening of the old year, and the snow was falling fast. In the cold and the darkness, a poor little girl, with bare head and naked feet, roamed through the streets. It is true she had on a pair of slippers when she left home, but they were not of much use. They were very large, so large, indeed, that they had belonged to her mother, and the poor little creature had lost them in running across the street to avoid two carriages that were rolling along at a terrible rate. One of the slippers she could not find, and a boy seized upon the other and ran away with it, saying that he could use it as a cradle, when he had children of his own. So the little girl went on with her little naked feet, which were quite red and blue with the cold. In an old apron she carried a number of matches, and had a bundle of them in her hands. No one had bought anything of her the whole day, nor had any one given her even a penny. Shivering with cold and hunger, she crept along; poor little child, she looked the picture of misery. The snowflakes fell on her long, fair hair, which hung in curls on her shoulders, but she regarded them not.

It was terribly cold and nearly dark on the last evening of the old year, and the snow was falling fast. In the cold and the darkness, a poor little girl, with bare head and naked feet, roamed through the streets. It is true she had on a pair of slippers when she left home, but they were not of much use. They were very large, so large, indeed, that they had belonged to her mother, and the poor little creature had lost them in running across the street to avoid two carriages that were rolling along at a terrible rate. One of the slippers she could not find, and a boy seized upon the other and ran away with it, saying that he could use it as a cradle, when he had children of his own. So the little girl went on with her little naked feet, which were quite red and blue with the cold. In an old apron she carried a number of matches, and had a bundle of them in her hands. No one had bought anything of her the whole day, nor had any one given her even a penny. Shivering with cold and hunger, she crept along; poor little child, she looked the picture of misery. The snowflakes fell on her long, fair hair, which hung in curls on her shoulders, but she regarded them not.

Figure 3: Parameters used in this experiment: `\pdfadjustlimit = 30`, `\tolerance = 500`, `\spaceskip = \fontdimen2\font plus\fontdimen3\font minus\fontdimen4\font`

- After breaking a paragraph into lines, we mark hboxes containing text lines created in this phase as boxes that might need adjustment for interword spacing.
- During glue setting of each hbox, we check whether it has been marked in the previous step. For every marked hbox we calculate the amount of font expansion for the box, depending on the sum of the character widths contained in the box and the amount of stretching/shrinking for the box.
- Finally, when shipping out marked boxes, we expand the font using horizontal scaling in PDF.

Thus the adjustment is applied only to those boxes that have been created when breaking paragraphs into lines. Also, we adjust only those boxes that need such correction: boxes with infinite stretchability and shrinkability are not changed.

There are two new primitives controlling this additional adjustment: a positive value of an integer parameter `\pdfadjustspacing` turns the adjustment on, and the value of an integer parameter `\pdfadjustlimit` specifies the limit of font expansion in thousandths of the original font width. For example, a value 50 of `\pdfadjustlimit` means that the font expansion must not exceed 5% of the original font width.

The font scaling is performed by changing the text matrix when needed. The drawback of this approach is that it makes the size of the PDF output larger. Displaying and printing such output files also takes more time.

Experimental Results

The text in the experiments was taken from the *The Little Match-Seller* by Hans Christian Andersen. All tests were run using the font *Utopia-Regular* at 11pt.

Parameters that were used to adjust line-breaking and interword spacing are indicated for each run. The left column is typeset normally, whilst the right column is typeset with font adjustment turned on. The common setting for all tests is `\frenchspacing`, `\hsize = 2.4 in` and `\emergencystretch = 1 em`.

Testing has suggested that the difference between the maximum values for stretching and shrinking should not exceed 50–60, otherwise the font scaling will be visible and give very ugly results. Thus `\pdfadjustlimit` should not be set to a value greater than 50 while adjusting paragraphs where all the interword spaces are either all shrunk or all stretched. In cases where interword spaces can be shrunk as well as stretched, the value of `\pdfadjustlimit` should not exceed 25–30.

The use of Multiple Master fonts might help by allowing the above limits to be exceeded; however we do not expect too much. In some cases where \TeX 's standard typesetting results were awful, the adjustment did not help very much.

Conclusion

Our experiments have shown that the “optimum fit” algorithm used in \TeX is much more powerful and useful than it might seem. The mechanism described above can be also used to achieve better typeset layout rather than to correct bad cases in an automatic way. Trying to adjust the interword spacing is not of much use in avoiding or reducing the need of for hand-tuning of line-breaking and spacing parameters. With some effort to establish appropriate values, it can considerably enhance the uniform level of overall greyness of the typeset results.

ltx2rtf: Exporting L^AT_EX documents to Word addicts

Daniel Taupin

Laboratoire de Physique des Solides
bt. 510, Centre Universitaire
F-91405 Orsay Cedex
France

Abstract

`ltx2rtf` is a compiler that translates L^AT_EX 2 ϵ source text into the RTF format used by several text processors, including Microsoft Word and Word for Windows. It was written by Fernando Dorner and Andreas Granzer in a one-semester course in Vienna (Austria) and is currently found as `latex2rtf` in CTAN servers.

It was heavily corrected and adapted to L^AT_EX 2 ϵ in 1997 by Daniel Taupin. The distribution was intended mainly for use within the MS-DOS window of Win95 and Win3.11, but all sources can be compiled on UNIX computers having GCC compilers.

Introduction: The need for a converter to RTF

Like most of the audience of TUG and other T_EXperts' meetings, I usually write most of my papers in L^AT_EX 2 ϵ . But problems arise when I need to transmit these documents to non-L^AT_EX users.

Various cul-de-sacs when transmitting L^AT_EX documents.

Transmitting a L^AT_EX document to other L^AT_EX users is no problem, since all L^AT_EX formats at least recognise the 7-bit representation of accented letters. The problem arises only when the addressee is reluctant to use a L^AT_EX representation: **Sending plain text.** This obvious (poor) solution fails because accented letters have at least three commonly used codings, the 850 for PCs, the Mac encoding and the ISO-latin1 coding, notwithstanding eastern European countries which use other ISO-8859 codings. Even the possible 7-bit bypass is often rejected since people who are not computer scientists seem allergic to the 7-bit representation “`r\’esum\’e`” instead of “`résumé`”.

Sending a PostScript file. This is apparently the “good” solution used by everyone in scientific areas. But it may fail for several reasons:

1. All *hard*-scientists have access to at least one PostScript printer, but administrative offices, as well as most private persons, may not due to the high cost of PostScript printers.
2. Even if they can access a PostScript printer, people receiving such a file in an e-mail under Windows have no standard means to send a PostScript file to their PostScript printer: as a matter of fact, Windows provides sev-

eral “drivers” for PostScript printers, but no driver which does nothing but plain transmission, which is possible only by using the UNIX `lp` or `lpr` commands, or the MS-DOS `copy` command.¹

3. Other software can solve this problem, but you cannot reasonably ask your correspondent (all of your correspondents in the case of a mailing list) to install either GhostScript, GhostView or `prfile10`.

Sending image files. One could think of sending images of the document, rather than the text with its layout. In fact, this is rather satisfactory if the document is one or two pages: a scanner can be used to produce GIF files, or several packages are available to help the knowledgeable producer of a complex document in converting it from DVI, PostScript, PCL to GIF, a format whose advantage is being compressed. However:

1. not all addressees are aware that they could use their Netscape or Microsoft Explorer to view a local GIF file; and
2. the GIF-bitmap file of a pure text page consumes much more space than the text it contains: the size is no problem for a few pages, but it is for dozens.

How to think of portability. The sender of a L^AT_EX document – as well as the sender of a C or F77 source program – is therefore faced with a *portability* problem.

Unfortunately, the person who exposes these difficulties is likely to get answers of the form: “Why

¹ Which does not work with network connected printers.

ltx2rtf : exporting L^AT_EX documents to Word addicts

Daniel TAUPIN
laboratoire de Physique des Solides
bât. 510, centre universitaire
F-91405 Orsay Cedex

July 18, 1998

Abstract

`ltx2rtf` is a compiler that translates L^AT_EX 2_ε source text into the RTF format used by several text processors, including Microsoft Word and Word for Windows. It was written by Fernando DORNER and Andreas GRANZER in a one-semester course in Vienna (Austria) and is currently found as `latex2rtf` in CTAN servers.

It was heavily corrected and adapted to L^AT_EX 2_ε in 1997 by Daniel Taupin. The distribution was intended mainly for use within the MS-DOS window of Win95 and Win3.11, but all sources can be compiled on UNIX computers having GCC compilers.

1 Introduction: the need for a converter to RTF

Like most of the audience of TUG and other T_EXperts' meetings, I usually write most of my papers in L^AT_EX 2_ε. But problems arise when needing to transmit these documents to non-L^AT_EX users.

1.1 The various cul-de-sacs when transmitting L^AT_EX documents

1.1.1 Sending plain text

This obvious (poor) solution fails because accented letters have at least three commonly used codings, the 850 for PCs, the Mac encoding and the ISO-latin1 coding, notwithstanding eastern European countries which use other ISO-8859 codings. Even the possible 7-bit bypass is often rejected since people who are not computer scientists seem allergic to the 7-bit representation “`rx\examm\ex`” instead of “`rxsumm`”.

1.1.2 Sending a PostScript file

This is the apparently “good” solution used by everyone in scientific areas. But it may fail for several reasons:

- All *hard*-scientists have access to at least one PostScript printer, but administrative offices, as well as most private persons, do not due to the high cost of PostScript printers.
- Even if they can access a PostScript printer, people receiving such a file in an e-mail under Windows have no standard means to send a PostScript file to their PostScript printer: as a matter of fact, Windows provides several “drivers” for PostScript printers, but no driver doing nothing except plain transmission which is possible only by using the UNIX `lp` or `lpr` commands, or the MS-DOS `copy` command¹.
- Other software can solve this problem, but you cannot reasonably ask your correspondent (all of your correspondents in the case of a mailing list) to install either GhostScript, GhostView or `prfile10`.

¹Which does not work with network connected printers.

1

2 The idea of ltx2rtf: using Word as a DVI driver

When sending a document to a variety of addressees, one should think of which software is most widespread among them; the answer is “thanks to Microsoft's powerful advertisements, they all possess² a version of Word[perfect] which can read RTF files.

Anyway, even with its deficiencies, `ltx2rtf` produces a RTF file which is quite satisfactory in the sense that it can be processed using Word, and nicely printed after several manual corrections, without the need to retype the whole of the text and add the font changes.

3 ltx2rtf

3.1 Features

In the same way as `latex2html`, `ltx2rtf` compiles the L^AT_EX source and directly produces RTF output, instead of HTML.

- Part, chapter and section numbering all use Word (and RTF) built-in macros to provide section numbers which can be updated when inserting new sections (i.e. “title” levels) as provided by Word. Optionally, these numberings can be computed by `ltx2rtf` itself, in such a way they are frozen for further Word updates.
- Conversely, `enumerate` environments produce frozen numbers, mainly because Word's built-in features inhibit unnumbered paragraphs within that environment.
- Western European accented letters – including capitals – are correctly treated, including the famous ISO-latin1 excluded “`œ`”, the german es-zet “`ß`”, the latin “`æ`”, the scandinavian “`ö`” and the spanish “`ñ`” (as well as `Ñ`).

But the deficiencies of our Word 6.0 make environment `multicol` unreliable.

3.2 Implementation

3.2.1 Maths handling

3.2.2 Quality of the result

From the L^AT_EX-er's view point, output (text and moreover maths) is much better than the results obtained by average Wordists³, especially with respect to lists. For example the famous formula using the discriminant $\sqrt{b^2 - 4ac}$ of the second degree equations:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

3.3 Availability

Software can be obtained from:

`ftp://ftp.lps.u-psud.fr/pub/ltx2rtf/ltx2rtf.zip`

4 Conclusion

In the same way that DVIPS is not intended to help typesetters moving from L^AT_EX to PostScript, `ltx2rtf` is not intended to help them moving from L^AT_EX to Word, but to help them in sending or posting nicely typeset papers, thus multiplying by tens the number of persons able to display and print it on their own Microsoft-addicted devices.

²Whether they actually bought the license is the addressee's problem, not mine.

2

Figure 1: Example L^AT_EX document.

don't you get rid of your Windows system and move to UNIX?”, “Why don't you discard your old Epson printer and have a PostScript printer?”, “Why don't you move from Microsoft's text editors and use L^AT_EX?”, “Why don't you install GhostScript, GhostView, `prfile10`, or a Linux partition to your PC?”, etc.

All these common sense answers are right, but they just forget one thing: the problem is not with *my* personal installation when sending/ mailing a document, the problem is with the installation of the addressees, whose skill I perhaps do not know at all, and who are probably unable to install software other than what they got when buying their personal computer or when registering on some multi-user workstation.

The idea of ltx2rtf: Using Word as a DVI driver

When sending a document to a variety of addressees, one should think of which software is most widespread among them; the answer is “Thanks to Mi-

crosoft's powerful advertisements, they all possess a version of Word[perfect] which can read RTF files.²

In fact, whatever many people claim about Microsoft's way of managing its software, RTF specifications are published by this company, and are available at: `ftp://ftp.microsoft.com/Softlib/MSLFILES/GC0165.EXE`, a self-extracting zipped file yielding a `*.DOC` file.³ Thus, using this specification file (130 pages) and testing the actual behaviour of Word,⁴ one can obtain a means of producing RTF from a L^AT_EX source.

This was attempted in 1994 by students at an institution which appears to be a Technical University in Vienna (Austria) and widely posted on CTAN under the name `latex2rtf`. Their translator is provided as several C source files which can be easily compiled with a satisfactory “makefile”.

² Whether they actually bought the license is the addressee's problem, not mine.

³ Unzipping it seems however to fail since that last posting. No comment...

⁴ An old Word 6.0 did not exactly respect the specifications...

ltx2rtf : exporting L^AT_EX documents to Word addicts
 Daniel TAUPIN
 laboratoire de Physique des Solides
 bât. 510, centre universitaire
 F-91405 Orsay Cedex

Abstract

ltx2rtf is a compiler that translates L^AT_EX₂ source text into the RTF format used by several text processors, including Microsoft Word and Word for Windows. It was written by Fernando DORNER and Andreas GRANZER in a one-semester course in Vienna (Austria) and is currently found as *latex2rtf* in CTAN servers. It was heavily corrected and adapted to L^AT_EX₂ in 1997 by Daniel Taupin. The distribution was intended mainly for use within the MS-DOS window of Win95 and Win3.11, but all sources can be compiled on UNIX computers having GCC compilers.

1 Introduction: the need for a converter to RTF

Like most of the audience of TUG and other T_EXperts' meetings, I usually write most of my papers in L^AT_EX₂. But problems arise when needing to transmit these documents to non-L^AT_EX users.

1.1 The various cul-de-sacs when transmitting L^AT_EX documents

1.1.1 Sending plain text

This obvious (poor) solution fails because accented letters have at least three commonly used codings, the 850 for PCs, the Mac encoding and the ISO-latin1 coding, notwithstanding eastern European countries which use other ISO-8859 codings. Even the possible 7-bit bypass is often rejected since people who are not computer scientists seem allergic to the 7-bit representation "r\oeum" instead of "rsame".

1.1.2 Sending a PostScript file

This is the apparently "good" solution used by everyone in scientific areas. But it may fail for several reasons:
 1. All *hard*-scientists have access to at least one PostScript printer, but administrative offices, as well as most private persons, do not due to the high cost of PostScript printers.
 2. Even if they can access a PostScript printer, people receiving such a file in an e-mail under Windows have no standard means to send a PostScript file to their PostScript printer: as a matter of fact, Windows provides several "drivers" for PostScript printers, but no driver doing nothing except plain transmission which is possible only by using the UNIX `lp` or `lpr` commands, or the MS-DOS `copy` command.
 3. Other software can solve this problem, but you cannot reasonably ask your correspondent (all of your correspondents in the case of a mailing list) to install either GhostScript, GhostView or `psfile10`.

¹ Which does not work with network connected printers.

2 The idea of ltx2rtf: using Word as a DVI driver

When sending a document to a variety of addressees, one should think of which software is most wide-spread among them; the answer is "thanks to Microsoft's powerful advertisements, they all possess² a version of Word(perfect) which can read RTF files. Anyway, even with its deficiencies, *latex2rtf* produces a RTF file which is quite satisfactory in the sense that it can be processed using Word, and nicely printed after several manual corrections, without the need to retype the whole of the text and add the font changes.

3 ltx2rtf

3.1 Features

In the same way as *latex2html*, *ltx2rtf* compiles the L^AT_EX source and directly produces RTF output, instead of HTML.
 - Part, chapter and section numbering all use Word (and RTF) built-in macros to provide section numbers which can be updated when inserting new sections (i.e. "title" levels) as provided by Word. Optionally, these numberings can be computed by *ltx2rtf* itself, in such a way they are frozen for further Word updates.
 - Conversely, *enumerate* environments produce frozen numbers, mainly because Word's built-in features inhibit unnumbered paragraphs within that environment.
 - Western European accented letters – including capitals – are correctly treated, including the famous ISO-latin1 excluded "œ", the german es-zet "ß", the latine "æ", the scandinavian "ø" and the spanish "ñ" (as well as Ñ). But the deficiencies of our Word 6.0 make environment `multicol` unreliable.

3.2 Implementation

3.2.1 Maths handling

3.2.2 Quality of the result

From the L^AT_EX'er's view point, output (text and moreover maths) is much better than the results obtained by average "Wordists", especially with respect to lists. For example the famous formula using the discriminant $\sqrt{b^2 - 4ac}$ of the second degree equations:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

3.3 Availability

Software can be obtained from:
<http://ftp.lps.u-psud.fr/pub/ltx2rtf/ltx2rtf.zip>

4 Conclusion

In the same way that DVIPS is not intended to help typesetters moving from LaTeX to PostScript, *ltx2rtf* is not intended to help them moving from L^AT_EX to Word, but to help them in sending or posting nicely typeset papers, thus multiplying by tens the number of persons able to display and print it on their own Microsoft-addicted devices.

² Whether they actually bought the license is the addressee's problem, not mine.

Figure 2: Example output from *latex2rtf*.

The C coding is clean and well structured but, unfortunately, the students did not have a knowledge of L^AT_EX of the same quality as their C programming skill; thus many things had to be revised concerning font management, sectioning, `itemize`, `enumerate`, `description` and `tabular` environments, notwithstanding L^AT_EX₂ε more recent specifications.

Anyway, even with its deficiencies, *latex2rtf* produces a RTF file which is quite satisfactory in the sense that it can be processed using Word, and nicely printed after several manual corrections, without the need to retype the whole of the text and add the font changes.

ltx2rtf

Features. In the same way as *latex2html*, *ltx2rtf* compiles the L^AT_EX source and directly produces RTF output, instead of HTML.

Part, chapter and section numbering all use Word (and RTF) built-in macros to provide section numbers which can be updated when inserting new sections (i.e. "title" levels) as provided by Word. Optionally, these numberings can be computed by *ltx2rtf* itself, in such a way that they are frozen

for further Word updates. Conversely, `enumerate` environments produce frozen numbers, mainly because Word's built-in features inhibit unnumbered paragraphs within that environment.

Western European accented letters – including capitals – are correctly treated, including the famous ISO-latin1 excluded "œ". In the same way, additional abbreviation features provided by Bernard Gaule's `french.sty` and Daniel Taupin's `smallcap.sty` (which enables a `\scfamily` command instead of `\scshape` to provide bold and/or slanted small capitals).

Implementation. Basic.

1. The input code can be either 7-bit, or ANSI (ISO-latin1) or 850. The Mac coding is not yet implemented but doing that would not be a problem.
2. The source can be compiled with any GCC compiler (no serious problems with other normal C compilers). We tested it mainly with the DJGPP port of GCC to DOS (native, Win3.11 and Win95).

3. Nothing more is needed, as long as one does not want to translate maths.
4. Maths are tentatively translated using the few RTF mathematical features such as raising parts of the text and changing fonts (size and shape).

Maths handling. Two options are provided for maths handling.

1. The `-m` option uses L^AT_EX-ing for *displayed equations*, namely those enclosed with `$$` (equation environment in the future). Then, nearly in the same way as `latex2html`:
 - `ltx2rtf` calls `latex` to produce a DVI file for each equation;
 - `ltx2rtf` calls an external procedure (DVI2PBM.BAT under MSDOS) which, in turn, either calls emTeX's `dvidrv dvidot` to produce PCX files and then calls NETPBM routines to convert the PCX to PBM, or calls DVIPS to produce a PostScript file and then GhostScript to produce a PBM file;⁵ and
 - finally, the PBM file is read by `ltx2rtf` itself and converted to “wbimap” as specified in the RTF specification document.⁶
2. The `-M` option not only uses L^AT_EX-ing for *displayed equations*, but also for single `$`-enclosed mathematical text.

Quality of the result. From the L^AT_EX-er's view point, output (text and moreover maths) is much better than the results obtained by average ‘Wordists’, especially with respect to lists.

Therefore the RTF produced is very good when one wants to e-mail a L^AT_EX-typeset text to unknown (or known) addressees whose probability of possessing Word is 95%, but of having at least DVI printers/viewers or easy access to PostScript printers is only 5%.

The inconveniences. From the producer's view-point, one sees the same installation difficulties as with `latex2html` with the exception that neither Perl nor GDBM/DBM are needed.

But more major inconveniences are seen from the addressee's viewpoint:

- Since Microsoft now rules the computation and networking world, any apparent deficiency in its products must be considered as “not a bug,

but a feature”. Therefore, any layout different from what a Wordist usually gets (think of no paragraph hanging indentation in hierarchical lists) may be considered as a negative feature, in the same way as accented capitals or non-english characters, which are so difficult to type (4 clicks and 3 mouse moves to produce æ or œ in French).

- Even worse, the L^AT_EX-like layout takes advantage of the powerful basic commands of RTF – something near to T_EX primitives plus some plain-T_EX facilities – but such results are nearly impossible to obtain with Word's ready-made clicking commands.

The reason is that the `ltx2rtf`-generated format (“format” in the Word sense) is definitely different from those which are provided as standard in Word's clicking windows. This results in the impossibility for the addressee to modify the RTF file except for pure text corrections and, perhaps, changes in sectioning (but not in section numbering style).

- Obviously, math parts are frozen as images which can only be removed, moved, enlarged or shrunk, but not edited.

Availability. The software can be obtained from: <ftp://ftp.lps.u-psud.fr/pub/ltx2rtf/ltx2rtf.zip>.

Conclusion

In the same way that DVIPS is not intended to help typesetters moving from LaTeX to PostScript, `ltx2rtf` is not intended to help them moving from L^AT_EX to Word, but to help them in sending or posting nicely typeset papers thus multiplying by tens the number of persons able to display and print it on their own Microsoft-addicted devices.

⁵ Thanks to Emmanuel Bigler who provided this alternate solution.

⁶ Other picture specifications are described, but they all fail with Word 6.0; therefore we kept to the only one succeeding.

Adding Native Language Support to the CWEB package and the T_EX program

Włodek Bzyl

Instytut Matematyki, Uniwersytet Gdański, Wita Stwosza 57, 80–952 Gdańsk, Poland

matwb@univ.gda.pl

Abstract

By adding National Language Support (NLS, for short) to literate programs I propose making such changes in their text via change files, which make modified programs aware of and able to support multiple languages. This paper describes how the GNU *libc* and *gettext* libraries were used to add NLS to the CWEB package and presents a possible way of bringing NLS to the T_EX program.

Introduction

In 1984 D. E. Knuth wrote about the WEB system [Literate Programming. *The Computer Journal*, 27]: “I made a conscious decision not to design a language that would be suitable for everybody. My goal was to provide a tool for system programmers, not for high school students or for hobbyists. I don’t have anything against high school students and hobbyists, but I don’t believe every computer language should attempt to offer all things to all people.”

Now, it can be said that WEB systems are used by a small elite of literate programmers who are able to combine their English verbal and programming skills. This is so, because all existing WEBS have been created with English conventions in mind, so these tools should not be expected to work well in non-English environments. Having realized that, I asked myself: does a WEB system exist that could be untied from the English conventions and tied anew to conventions of another language? At that time I was experimenting with FWEB, *noweb*, and CWEB. Of these systems only CWEB supported the use of non-Latin characters in literate programs. This feature made me believe that a CWEB adaptation to the Polish conventions and the conventions of other languages is possible. After some work I had a version of CWEB usable by any programmer literate in Polish.

When I was experimenting with the changed CWEB, new GNU *libc* and *gettext* libraries appeared. These libraries make it possible to write C programs that automatically adapt to local sets of conventions set up by the values of some environment variables. This forced me to rethink what I had done. New functionality offered by functions from these libraries makes it possible to

have *one* CWEB that could be used for writing literate programs in English, Polish and many other languages.

If this idea is feasible, it would make literate programming accessible to programmers who like to write and to explain what they are doing in the language of their choice, or in the language appropriate for the audience to whom they are going to present their concepts and ideas. Moreover, the World Wide Web would not be populated with slightly different CWEBs and the CWEB system of Levy and Knuth will stay open for improvements by everyone.

Programming interface for NLS — the ISO C model

In the ISO C model, NLS works by means of *locales* divided into six categories, to be selected and activated independently. Each category specifies a collection of conventions — one set of conventions for each category. Here is the list of all categories.

- LC_CTYPE — specifies the character set
- LC_COLLATE — specifies the conventions for sorting order
- LC_MESSAGES — specifies the language for messages
- LC_MONETARY — specifies the formatting of monetary quantities
- LC_NUMERIC — specifies the formatting of numbers
- LC_TIME — specifies the formatting of dates and times

Each category name is both a macro name to be used in C code and an environment variable that a user can set. There is also a special C macro LC_ALL used to select all sets of conventions and there are two special environment variables.

LC_ALL — if defined, its value specifies the locale to use for all purposes

LANG — if defined, its value specifies the locale to use for all purposes except as overridden by any of the variables above.

In C code, the `setlocale` function is the main means for specifying the categories to be used. It does not change the program behavior directly. Rather, the selected locale data is used by some functions from the C library. For example, the functions `strcoll` and `strxfrm` will use the sorting order defined in the Polish locale whenever the value of environment variable `LC_COLLATE` is `p1`. If the `LC_MESSAGES` value is `p1`, then the users will see Polish messages on their screen, supposing that a catalog of messages with the translations of the messages into Polish can be found.

According to the authors of the *gettext* manual, bringing NLS to a C program is an easy two step process.

1. [Internationalization.] Parameterize the program code so that it does not include specific cultural conventions in its output code and in its message strings.
2. [Localization.] Specify for each locality of users the set of cultural conventions and the catalog of message strings to be used by the program output code.

Although it could not be warranted that these two steps could be successfully performed on existing code, in the next section it will be shown how NLS can be added to the CWEB package. The paper concludes with remarks on a possible way of bringing NLS to the \TeX program.

Adding NLS to the CWEB package

Look for special macro packages designed for CWEB users in your language; or, if you are brave, write one yourself.

— CWEB user manual

The purpose of the current section is to propose a possible way of bringing NLS to the programs `ctangle`, `cweave`, and to the \TeX macro file `cwebmac.tex`; i.e., to the main components of the CWEB package. Out of several possible ways of doing that, I decided to use the ISO C model because of the existing support in the newest GNU *libc* and *gettext* libraries.

Clearly, the ISO C model could be used with the literate CWEB programs, because they are essentially C text. Therefore, a project of bringing

NLS to the CWEB package appears to be feasible, but a more detailed analysis is necessary.

Let us remember what the `ctangle/cweave` pair of programs actually does. In literate programming, `ctangle` creates a C program and `cweave` creates a `.tex` file. The first line of the produced `.tex` file tells \TeX to input the file `cwebmac.tex` with macros defining CWEB's documentation conventions. Finally `cweave` will generate a sorted cross-reference identifier index, alphabetized lists of the section names, and a table of contents. In case of errors, both programs send various clues about errors to the computer screen.

Recasting the above description in terms of the ISO model we get the following TODO list:

LC_COLLATE — the code responsible for sorting [§228–§239, `cweave.w`] should be changed. In particular, the collation mapping, based on the data read from the `LC_COLLATE` locale, should be created at runtime. A closer inspection of the code [§235, `cweave.w`] reveals that `LC_CTYPE` data should be used too, as the original collation mapping does not differentiate between uppercase and lowercase characters. This implies that the environment variables `LC_COLLATE` and `LC_CTYPE` should have the same value. Otherwise, we end up with a corrupted collation table with lowercase characters not mapped onto their uppercase equivalents. Therefore, we only read the value of the `LC_COLLATE` environment variable and use the value to read the `LC_CTYPE` locale data.

LC_MESSAGES — strings to be translated should be marked. Here, for each literate program, a separate change file should be created, with a code that initializes locale data and with strings to be translated being marked. Next, everything should be ctangled and the `xgettext` tool should be used to create an initial message catalog from the produced C sources.

LC_MONETARY, **LC_NUMERIC** — nothing to be done for these categories.

LC_TIME — the `\today` macro should be redefined. Otherwise, the file created by `cweave` and typeset by \TeX will contain English month names. This raises the following question: how to make the expansion of the `\today` macro depend on the values of environment variables, as they are at the time when the file is being made? This question is a particular case of a more general one: how to characterize the file produced by an internationalized `cweave`?

An admissible answer could be: the created file should be able to instruct \TeX which format to use for typesetting. Additionally, the file should

redefine the `cwebmac.tex` macros that output the English text together with the extra fonts being used.

It is a little known fact that the current Web2C implementation of T_EX makes it possible to choose the format at runtime with a `%&` line.¹ This feature appears to provide us with a way of producing such a file. It suffices that the first and second line of the file created by `cweave` are:

```
%&(LC_COLLATE variable value)
\input cweb-(value of LC_COLLATE variable).tex
```

Unfortunately, the first line could not be created automatically, because names of format files do not reflect the language supported by the format. For example, the name `mex` does not tell us that the `mex.fmt` is the adaptation of the `plain` format to Polish conventions. Even if it were possible to create the first line automatically, the file would be typeset incorrectly—almost all characters with codes from the 128–255 range would be missing, or they would be replaced improperly. This is due to the fact that the encoding used for writing down a file differs from the internal encoding of fonts used for typesetting. For example, to typeset correctly a Latin-2 encoded file written in Polish, the file should be presented to T_EX as PL-encoded, which is the internal font encoding used by the Polish Computer Modern fonts.

It should be noted, that some T_EX implementors have approached the problem of such a change of encoding. For example, Eberhard Mattes in his `emTeX` enables the user to save re-encoding mapping in format files. The `EncTeX` package (the Extension of T_EX for the Reencoding of the Input) by Petr Olšák provides new primitives to be used for creating re-encoding mapping to be saved in a generated format.

These non-standard extensions proliferate format files and make them depend on the file encoding. This is not necessary, because there are other ways of making re-encoding superfluous; for example, by instructing the current shell to do re-encoding. Alas, none of the shells known to the author allow such re-encoding.

Another possibility, based on a commented out TCX-code in `Web2C`, would be to add the following option to the `%&` line

```
-translate-file=(LC_COLLATE variable value)
```

¹ This notation is analogous to the `#!` notation used in shell scripts to tell the kernel which shell runs the script.

which tells `tex` to do input re-encoding defined in the `translate` file. Unfortunately, this line would be ignored by the `tex` program, as the user is not allowed to put anything on the `%&` line except just the format name.

As I am generally against the unnecessary proliferation of format files, yet another approach has been chosen. This approach does not yield, as described above, a self-contained file. The format and the re-encoding name must be written whenever `tex` is run. For example, the following command

```
tex -fmt=mex -translate-file=pl foo
```

initiates the typesetting of `foo.tex` with character codes re-mapped by a table read from the `pl.tcx` `translate` file. This command should be used to invoke `tex` on a Latin-2 encoded file written in Polish.

To get the behavior just described, it suffices to output the second line (from the two lines displayed on the left and above), where the file being input consists of two lines:

```
\input cwebmac.tex
\input (LC_COLLATE variable value)-cweb.tex
```

Here, the second input file should redefine the macros that output English text. Moreover, this file could be used to add and to redefine extra fonts, because all text fonts being used share the same internal encoding.

Literate programmers should have the option of using any 8-bit character code, even in identifiers of the C program. Because there is no internationalized C compiler around, which, by definition, would allow the use of all character codes as identifiers, the authors of CWEB made `ctangle` to recognize character codes from the forbidden range 128–255 and to replace them by strings read from a default, or user-constructed, transliteration table.

All the described changes have been implemented. The result is the CWEB-NLS package. The change files of the package could be used for converting `ctangle` and `cweave` into programs that easily adapt to local conventions. Switching between different languages is achieved by setting the `LANG` environment variable to the appropriate language prior to using internationalized programs. For example, let's presume that the Polish language is requested. At the shell prompt, or from the users' startup files the following command should be executed

```
export LANG=pl
```

(in bash, or an equivalent command in other shells). If users prefer to see English messages on their screen, they should execute

```
export LC_COLLATE=pl
```

and all NLS magic will happen automatically.

Let's conclude this section with some statistics. The CWEB-NLS package consists of 9 orthogonal change files, where each file implements a different functionality. These change files could be applied with other change files extending `ctangle` and `cweave` in other ways. The total number of changed sections, from the total of 406, is 110 (around 20, if not counting the trivial changes). The 112 different messages output by `ctangle` and `cweave` have been translated into Polish. Two \TeX macro files have been written to make possible a mechanical creation of localized macro files. Finally, 36 different strings output by the `cwebmac.tex` macros were translated and 7 fonts were changed too.

\TeX and NLS

In this section I am going to concentrate on the possible ways of implementing message catalogs. The proposed implementation applies also to METAFONT and METAPOST. This section will conclude with a list of proposed changes to the Web2C code to make it more NLS friendly.

“Free software is going international! The Free Translation Project is a way to get maintainers of free software, translators, and users all together, so that they will gradually become able to speak many languages.” — this is how the ABOUT-NLS file begins. I consider this GNU project very important for the reasons explained above and I am glad to see many GNU packages already speaking in Polish (see Appendix B for the current state of the project). But it came as a surprise to find traces of NLS support in Web2C. In the file `texmfmp.c`, which is a part of `tex`, `etex`, `pdftex`, `omega`, `mf`, `metapost` code, I found the statement `setlocale(LC_CTYPE, "")`. Unfortunately, this statement makes programs depend on the values of the `LC_CTYPE`, `LANG`, `LC_ALL` environment variables. In particular, the following C functions are affected: `isgraph`, `isspace`, `isprint`, `islower`, `isdigit`. Another unpleasant surprise came, when I executed the following command

```
LANG=pl tex -format-file=mex foo
```

It showed the following strange output

```
tex: nieznaną opcją '-format-file=mex'
Try 'tex -help' for more information.
```

— a mixture of Polish and English. Generally, I do not like to be surprised by software in this way. Therefore I decided to examine the code. \TeX is assembled from literate sources combined with various change files and hand-coded routines spread among several C source files. These files are not localized and the non-localized code is responsible for the appearance of the untranslated messages.

Unfortunately, the literate part of \TeX does all of its string processing by “home-grown methods” [38, `tex.web`] and string handling in \TeX could not be localized with the GNU tools because format files play a rôle of message catalogs. To see this, let's recall the relevant facts. All the strings output by `tex` are contained in the `tex.pool` file with a check sum appended at the end. The check sum replaces the place-holder `$@` in `tex.p`—the tangled Pascal source of `tex`. When the `tex` program is preparing itself to dump a format file it reads the strings from the pool file and writes these strings on the format file. The `tex` program reads these strings from format files. Whenever `tex` is run it examines the check sums and gives up when the check sums do not match. The above description shows that removing the messages from format files requires considerable changes in the code. For that reason, I propose another approach for how to handle the translated strings.

If we ignore the check sums, then it suffices to translate all the 1309 strings from the `tex.pool` file and to update the string lengths. Now, `tex.pool` with translated strings could be used to build a format with the command²

```
tex -ini -translate-file=pl mex
```

It is essential to use the translation file and to recode the strings into the internal encoding of the fonts being used. Otherwise, these strings would not be re-encoded correctly when written back onto the computer screen. It should be noted that the produced format does not depend on the encoding of the \TeX file. For example, it could be used to typeset CP852 encoded \TeX sources written in Polish. The actual command to be used should begin with

```
tex -fmt=mex -translate-file=cp852
```

where we assume that `cp852.tcx` file contains an appropriate translation table.

It is not particularly difficult to extend the above approach to preserve the examining of check

² Actually, due to the way TCX files are implemented, the pool file has to be translated to the internal font encoding by other means.

sums. This would require changes in three sections of `tex.web` (§53, §1307, §1308).

Let's conclude with the promised WISH LIST:

- extend the syntax of the `%&` line to allow including user options
- localize all C sources in the `Web2C` directory
- implement the mapping file, analogously to `texfonts.map`, which will allow one to rename format files; the file will allow automatic generation of fully internationalized files as described above.
- translate `tex.pool`

References

Drepper, Ulrich. "Internationalization in the GNU project", 1997.

Drepper, Ulrich, Jim Meyering and François Pinard. "GNU gettext tools", May 1998.

Knuth, Donald E. "Literate Programming", *The Computer Journal* **27** (1984), pp. 97–111.

Knuth, Donald E., and Silvio Levy. "The CWEB System of Structured Documentation", version 3.0. `cwebman.tex` file from the CWEB package. ABOUT-NLS, file available on most GNU archive sites.

Appendix A

Someone writes in `tex.ch`: „TCX files are probably a bad idea, since they make T_EX source documents unportable. Try the `inputenc` L^AT_EX package.”

I think that `inputenc` package does not provide the functionality offered by TCX files for the following reasons:

- the `\uppercase` primitive does not work
- commands names which use diacritical characters could not be defined
- the `inputenc` package is usable for L^AT_EX only—other formats are not supported (there are already 42 different formats on the T_EX Live 3 CD-ROM)
- the log file and terminal are not readable because unreadable ‘`^^`’ notation is used—see the example below.

```
Niewypelnione \hbox (lichość 10000)
znaleziono w linii 2
\rm kość
\hbox(6.88889+0.0)x44.0
.\rm k
.\kern-0.27779
.\rm o
.\rm ś
.\rm ć
```

[1])
Wyjście zapisane do `foo.dvi`

```
(1 strona, 212 bajtów)
Niewype^^c2nione \hbox (licho^^c9^^b8 10000)
znaleziono w linii 2
\tenrm ko^^b1^^a2
\hbox(6.88889+0.0)x44.0
.\tenrm k
.\kern-0.27779
.\tenrm o
.\tenrm ^^b1
.\tenrm ^^a2

[1] )
Wyj^^c9cie zapisane do foo.dvi
(1 strona, 212 bajt^^f3w)
```

Appendix B

The following matrix shows, for several countries, the current state of internationalization in the GNU project, as of May 1998. The following matrix shows, with regard to each package listed, the languages in which message catalogs have already been submitted.

Ready P0 files	cs	de	fr	nl	no	pl	ru	sl	sv
bash	▪	▪	▪						
bison	▪	▪	▪						
clisp	▪	▪							
cpio	▪	▪	▪			▪			
diffutils	▪	▪	▪			▪			▪
enscript	▪	▪	▪					▪	
fileutils	▪	▪	▪	▪		▪		▪	▪
findutils	▪	▪	▪	▪		▪	▪		▪
flex		▪							▪
gcal	▪	▪	▪			▪			▪
gettext	▪	▪	▪	▪	▪	▪	▪	▪	▪
grep	▪	▪	▪	▪	▪	▪	▪	▪	▪
hello	▪	▪	▪	▪	▪	▪	▪	▪	▪
id-utils	▪	▪							
indent	▪					▪	▪		
libc	▪	▪	▪			▪			▪
m4	▪	▪	▪				▪		▪
make	▪	▪	▪			▪			
music		▪							
ptx	▪	▪	▪	▪	▪	▪	▪		▪
recode	▪	▪	▪	▪	▪	▪	▪	▪	▪
sh-utils	▪	▪	▪	▪	▪	▪	▪		▪
sharutils	▪	▪	▪	▪	▪	▪	▪		▪
tar	▪	▪	▪	▪	▪	▪	▪		▪
texinfo	▪	▪	▪						
textutils	▪	▪	▪	▪	▪	▪	▪		▪
wdiff	▪	▪	▪	▪	▪	▪	▪		▪

Pretprin — a L^AT_EX 2_ε package for pretty-printing texts in formal languages

Marcin Woliński

wolinski@melkor.mimuw.edu.pl

Abstract

A L^AT_EX 2_ε package is presented which provides tools for building lexical and syntactical analyzers in T_EX that can be used for pretty-printing. Examples of such analyzers for the programming languages Pascal and Prolog are shown as well as a small example of a new analyzer definition.

The problem

In most books on computer science published today, algorithms are presented in a way known to every T_EXnician as *verbatim*. In some well printed books however programs in algorithmic language Pascal are formatted in a rather complicated way, with bold keywords, italic identifiers and appropriate indentation on every line:

```
begin q := 0; r := x;  
while r ≥ y do  
  begin r := r - y; q := q + 1  
  end  
end
```

Such a layout gives a graphical representation for logical structure of the program. Indentation shows control flow: one can easily see “which **end** matches which **begin**”. Keywords are put in boldface while other identifiers are in italic, so even a beginner is able to recognize which elements are predefined parts of the language. Every occurrence of a given structure is formatted the same way, making it easier to “navigate” in the text.

This paper is dealing with the question how to generate such a layout with T_EX.

A large amount of tedious work would be needed to add all typesetting commands by hand. Moreover it would be difficult to get a consistent result in, e.g., a 200-page book.

Fortunately someone already had this problem before. The person was Prof. Knuth, who wanted to publish a few large programs written in Pascal including T_EX and METAFONT. Since he wanted to get good quality printouts in a finite length of time, he decided to teach the computer how to typeset Pascal. This way of generating pretty-printed text of the program became one of the WEB system functions.

The WEB system (or rather, WEAVE, which deals with the processing of the program’s documentation) performs syntactical analysis to recognize language constructs such as **if** — **then** — **else**, **repeat** — **until**, etc. This provides a layout consistency that is hard to achieve with other techniques.

WEB is a really smart tool for generating technical documentation for programs. But using WEAVE to process a book on computer science, containing some random pieces of code that are not supposed to build a working program, is somewhat unnatural. In such a case it would be preferable to avoid using external tools. But that would mean teaching T_EX itself how to pretty-print Pascal.

That is precisely what Pretprin was meant to do. The source for the example above in Pretprin’s notation is:

```
\begin{Pascal*}  
  begin q: =0;r: =x;while r>=  
    y do begin r: =r-y;q: =q+1end end  
\end{Pascal*}
```

One more insight was crucial for Pretprin’s architecture: in WEAVE (being a Pascal program), the rules of parsing Pascal texts are expressed with a series of complicated **if** — **then** — **else** constructs. In T_EX, on the other hand, it’s much easier to write a general interpreter for such rules. But this means language-specific rules are separated from the rest of the program and it is relatively easy to substitute them with other sets of rules.

So in its present shape Pretprin is mainly a toolbox for building scanners and parsers in T_EX. The tools can be used to analyze any sufficiently regular data, which probably includes any commonly used formal language.

Examples of Pretprin usage

Pretprin is loaded with `\usepackage` commands stating which language-specific modules should also be loaded. For example, to typeset a document containing pieces of Pascal and Prolog use:

```
\usepackage[pascal,prolog]{pretprin}
```

The Pascal module provides a L^AT_EX environment named `Pascal` that is used to typeset displayed pieces of Pascal programs:

```
\begin{Pascal}
  var i,L,w:integer; ch:char;Z:
  array [1..wmax]of char;
  begin L:=0;repeat w:=0;read(ch);
  while (ch<>' ') and (ch<>eol)
  do ...
  until eof (input)
  end.
\end{Pascal}
```

Note in the output shown below that spacing, indentation and line-breaking was done by `Pretprin`, completely ignoring the layout of the input (which, by the way, is definitely bad). Moreover, relational symbols such as `=`, `<` or `≠` (which is substituted for `<>`), and binary operators like `+`, `..`, and `∧` (which replaces `and` in the source), are put into T_EX's math mode with correct spacing. The overall layout is very similar to that generated by `WEAVE`.

```
var i,L,w: integer; ch: char;
      Z: array [1 .. wmax] of char;
begin L := 0;
repeat w := 0; read(ch);
  while (ch ≠ '␣') ∧ (ch ≠ eol) do
    begin w := w + 1; Z[w] := ch;
         read(ch)
    end;
if w > 0 then
  begin if L + w < Lmax then
        begin write('␣'); L := L + 1
        end
      else begin write(eol); L := 0;
            end;
    for i := 1 to w do write(z[i]);
    L := L + w
  end
until eof(input)
end.
```

Inline pieces of code such as “Consider *A* being an `array [1 .. N] of integer...`” can be written as `Consider \pascal{A} being an \pascal{array [1..N] of integer}...`

The `Pretprin` module for pretty-printing Prolog (in modern syntax) defines an environment `Prolog`

and a command `\prolog` with one parameter. Here is an example of four Prolog clauses:

```
\begin{Prolog}
d(X,X,D):-atomic(X),!,D=1.
d(C,X,D):-atomic(C),!,D=0.
d(U+V,X,DU+DV):-d(U,X,DU),d(V,X,DV).
d(U*V,X,DU*V+U*DV):-d(U,X,DU),d(V,X,DV).
\end{Prolog}
```

The output is simpler than in the Pascal case, since Prolog's syntax is more terse. However, typographical symbols are substituted for `:-` and reasonable spacing is added.

```
d(X, X, D) ← atomic(X), !, D = 1.
d(C, X, D) ← atomic(C), !, D = 0.
d(U + V, X, DU + DV) ←
  d(U, X, DU), d(V, X, DV).
d(U * V, X, DU * V + U * DV) ←
  d(U, X, DU), d(V, X, DV).
```

Our last example shows a few rules of Stanisław Szpakowicz's formal grammar of a large subset of Polish. The notation used is a variation on the DCG (Definite Clause Grammar) theme.

Nonterminals in the grammar are put in bold, but conditions (marked with minus sign in front) are in normal weight. In arguments, variables are set in italic and constants in upright shape. The pretty-printer prefers to break lines between a nonterminal and a condition rather than between two conditions or two nonterminals, so conditions tend to group on separate lines. `Pretprin` also carefully takes into account the space needed by rule numbers on the right side of the column.

```
ZDANIETO
= ZDANIEOGR(NR, R, L, O, war, prze, NEG) (zt1)
= SPOJNIK(TO) (zt2)
ZDANIEOGR(nr, r, l, o, WAR, PRZE, neg) .

SZDRZ(p, r, MNO, o)
= SPOJLEWY(nr) -ALT(nr, 1.2.3) (szdrz1)
FRZ(p, r1, l1, o1) PRZEC
SPOJPRAWY(nr) FRZ(p, r2, l2, o2)
-UZGR(r1, r2, r) -MIN(o1, o2, o)
= FRZ(p, r1, l1, o1) PRZEC KSPOJ(wz) (szdrz2)
-ALT(wz, (A.TAKŻE).(JAK.RÓWNIEŻ)
.(JAK.TEŻ)) FRZ(p, r2, l2, o2)
-UZGR(r1, r2, r) -MIN(o1, o2, o) .
```

One more thing worth emphasizing here is that it is possible to use multiple `Pretprin` modules in a single document. For example, the current paper contains examples in three different programming languages and it was generated with a single L^AT_EX run.

How to build a pretty-printer

In this section we will try to build a pretty-printer for a very simple language of terms. Terms are abstract expressions which logicians and computer scientists just love to write. We will consider a basic notation for terms which allows atoms (names built from letters) and functors (having a name and, in parentheses, a list of arguments, each being a term). Here is a typical example of a term:

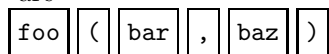
loves(John, and(Mary, and(Tom, Jerry)))

(No semantic interpretation please, terms are merely abstract structures.)

Lexical analysis (scanning). The first task in analyzing a string is to detect “words” in it. For the string

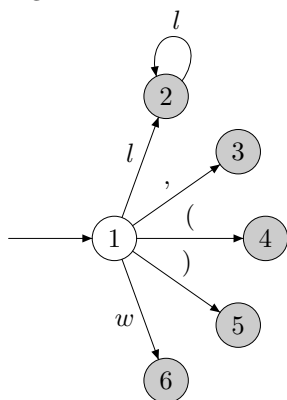
foo(bar , baz)

the “words” are



(Note that space characters were ignored.) We can distinguish here four “parts of speech”: sequences of Latin letters (atoms), opening parentheses, closing parentheses, and commas.

For splitting strings of characters into words we use an apparatus known as a finite state automaton. The diagram below represents an automaton for the example language:



In the example language the function of any letter is the same: letters are building-blocks for atoms. For that reason the notion of character groups is used. The following declarations define in \TeX parlance character groups l and w :

```
\DeclareGroup{l}
  {abcdefghijklmnopqrstuvwxyz}
\DeclareGroup{w}{ ^I^M}
\CompileGroups
```

Therefore the label l in the diagram denotes any letter, while the label w denotes any “white character”.

Let us now trace what happens when the string

foo(bar , baz)

is being run through our automaton. The automaton starts in state 1 and encounters the letter f from the input string. Since this character belongs to the “letters” group, the automaton performs a transition to state 2 (remembering the f). Now, the letter o is input, and since there is a transition from 2 to 2, reading a letter, the automaton does just that. Then another o comes that is handled similarly. And now, still in state 2, the automaton sees a $($. Since there is no arrow from 2 labeled with $($, the automaton cannot consume it and stops. State 2 is marked with a gray circle meaning that it is an accepting state; stopping in this state means a word has just been read. In our case, the word is foo . The action associated with state 2 will pass this word to the next processing stages.

Now a search is started for another word, so the automaton returns to state 1. A transition labeled with $($ leads to state 4. The next character is a space and there is no transition from 4 labeled with a space. The automaton stops, and a single parenthesis is recognized as the second word.

The next “word” consists of a single space. It is accepted in state 6, which is somewhat special in that the action associated with it is empty. This way spaces get gobbled.

The process continues until the whole string is processed.

Before we actually describe this automaton in \TeX we’ll take a closer look at the actions associated with states. These actions prepare a list of “scraps” on which the syntax analysis will work. This list is constructed with \AppendElem procedure. Every scrap has a grammatical category and translation (the actual text). These two constitute arguments of \AppendElem . In our case, categories are just the parts of speech mentioned earlier: atom, open, close and comma.

The first state is not accepting, so there is no action associated with it. The state has five transitions:

```
\DeclareTransition 1-l->2.
\DeclareTransition 1-,->3.
\DeclareTransition 1-(->4.
\DeclareTransition 1-)->5.
\DeclareTransition 1-w->6.
```

State 2 on the other hand is accepting, words of the category atom are recognized in it:

```
\DeclareState{2}{\AppendElem{atom}{#1}}
\DeclareTransition 2-l->2.
```

(#1 above is the string read as the automaton was going from the start state.) The rest of the states

have no leaving transitions but are accepting. Commas and parentheses are recognized in them, and in state 6, blank characters are gobbled.

```
\DeclareState{3}{\AppendElem{comma}{#1}}
\DeclareState{4}{\AppendElem{open}{#1}}
\DeclareState{5}{\AppendElem{close}{#1}}
\DeclareState{6}{}

```

And the last declaration specifies the starting state:

```
\CompileScanner{1}

```

Syntax analysis (parsing). Now the input string has been read and converted to the form

atom	open	atom	comma	atom	close
foo	(bar	,	baz)

The next task is to recognize the syntactical structure of the text. We already know the parts of speech, but now higher level grammatical categories can emerge. This process is described with a set of simple grammatical rules.

Our first observation will be that when an atom is immediately followed by an open parenthesis it is the beginning of a term. We'll call such an entity `termhd` (a term head):

atom open → termhd

In our example this rule allows us to derive that `foo(` is the beginning of a term.

If there is no parenthesis after an atom it surely is a term all by itself (this is the case with `bar` and `baz` in the example).

atom → term

Terms can have arguments, so the next rule describes how a `termhd` can grow: adding a term and a comma to a `termhd` gives another well formed `termhd`.

termhd term comma → termhd

And finally when after a `termhd` comes a term (the last argument) and a closing parenthesis the whole fabric can be stuffed into a new term:

termhd term close → term

(Note that we do not accept `foo()` as a term.)

These rules again in the T_EX notation are:

```
\DeclareProduction{atom,open}
  \ThisElem\TwoElems{termhd}
  {#2#1}\ThisElem
\DeclareProduction{atom}
  \ThisElem\OneElem{term}
  {\textit{#1}}\PrevElem
\DeclareProduction{termhd,term,comma}
  \ThisElem\ThreeElems{termhd}
  {#3\formatter{#2}#1\ } \ThisElem
\DeclareProduction{termhd,term,close}

```

```
\ThisElem\ThreeElems{term}
  {#3\formatter{#2}#1}\PrevElem

```

This notation is somewhat more verbose and allows us to describe situations where not all elements of the left side of a rule are to be collapsed into a new scrap (context rules).

The first rule can be read as follows: if you are looking at the `atom` scrap followed by an `open` scrap, do as follows: starting from `ThisElement` (the `open` scrap), take `TwoElements` and replace them with a scrap of category `termhd`, the translation of which was formed from the translations of `atom` (`#2`) and `open` (`#1`) scraps. Then continue the process starting from `This` (the inserted) `Element`.

The pretty-printing depends on building appropriate translations of complex grammatical entities. We have ignored all spaces in the input, so now we are responsible for putting them back in a consistent manner. Moreover, in the example language when an `atom` is being recognized as a term without arguments, `\textit` is applied to render the atom's name in italics. And whenever a term is added to the list of arguments of a term-under-construction, the macro `\formatter` is applied. In the definition of this macro we decide what it means to pretty-print a term. To keep things simple, we will just put a frame around each sub-term:

```
\newcommand\formatter[1]{\fbox{#1}}

```

With these definitions

```
\begin{Terms}
loves(John, and(Mary, and(Tom, Jerry)))
\end{Terms}

```

yields

loves(*John*, and(*Mary*, and(*Tom*, *Jerry*)))

Marcin Woliński

A Appendix

Components of arrays need not be scalars—they themselves may be structured. If they are again arrays, then the original array A is called *multidimensional*. If the components of the component arrays are scalars, then A is called a *matrix*. The declaration of a multidimensional array variable follows the pattern formulated in (11.1). For example, in the declaration

var M : **array** [$a \dots b$] **of** **array** [$c \dots d$] **of** T (11.26)

M is declared to consist of $b-a+1$ components (often called matrix rows) with indices a, \dots, b , each of which is an array of $d-c+1$ components of type T with indices c, \dots, d . To denote the i th component (matrix row) of M , the conventional notation

$$M[i] \quad a \leq i \leq b \quad (11.27)$$

is used, and its j th component of type T is denoted by

$$M[i][j] \quad a \leq i \leq b, \quad c \leq j \leq d \quad (11.28)$$

It is customary and convenient to use the following abbreviations, which are entirely equivalent to (11.26) and (11.28), respectively.

var M : **array** [$a \dots b, c \dots d$] **of** T (11.29)
 $M[i, j]$

Example: Multiplication of matrices. Given the two real-valued matrices $A(m \times p)$ and $B(p \times n)$ compute the matrix product $C(m \times n)$, as defined by

$$C_{ij} = \sum_{k=1}^p A_{ik} * B_{kj} \quad (11.30)$$

for $i = 1, \dots, m$ and $j = 1, \dots, n$.

The formulation of program (11.31) follows from (11.30) in a straightforward manner.

```
var  $A$ : array [ $1 \dots m, 1 \dots p$ ] of real; (11.31)
   $B$ : array [ $1 \dots p, 1 \dots n$ ] of real;
   $C$ : array [ $1 \dots m, 1 \dots n$ ] of real;
   $i$ :  $1 \dots m$ ;  $j$ :  $1 \dots n$ ;  $k$ :  $1 \dots p$ ;  $s$ : real;
begin { assignment of initial values to  $A$  and  $B$  }
for  $i$  :=  $1$  to  $m$  do
  for  $j$  :=  $1$  to  $n$  do
    begin  $s$  :=  $0$ ;
    for  $k$  :=  $1$  to  $p$  do  $s$  :=  $s + A[i, k] * B[k, j]$ ;
     $C[i, j]$  :=  $s$ 
    end
  end
end.
```

Figure 1: A page from chapter 11 of Niklaus Wirth's *Systematic Programming: An Introduction*

```

\documentclass{book}
\usepackage{pascal}

\begin{document}
...

Components of arrays need not be scalars---they themselves may be structured. If they are
again arrays, then the original array \pascal{A} is called \emph{multidimensional}. If
the components of the component arrays are scalars, then \pascal{A} is called a
\emph{matrix}. The declaration of a multidimensional array variable follows the pattern
formulated in (\ref{arrtype}). For example, in the declaration
\begin{Pascal}
  var M: array[a..b]of array[c..d]of T
\end{Pascal}\pplabel{abcdarray}
\pascal{M} is declared to consist of $b-a+1$ components (often called matrix rows) with
indices $a,\ldots,b$, each of which is an array of $d-c+1$ components of type \pascal{T}
with indices $c,\ldots,d$. To denote the $i$th component (matrix row) of \pascal{M}, the
conventional notation
\begin{equation}
  \pascal{M[i]}\quad a\leq i \leq b
\end{equation}
is used, and its $j$th component of type \pascal{T} is denoted by
\begin{equation}\label{ijthelem}
  \pascal{M[i][j]}\quad a \leq i \leq b, \quad c \leq j \leq d
\end{equation}

It is customary and convenient to use the following abbreviations, which are entirely
equivalent to (\ref{abcdarray}) and (\ref{ijthelem}), respectively.
\begin{equation}
  \begin{tabular}{t}{l}
    \pascal{var M: array [a..b,c..d] of T}\
    \pascal{M[i,j]}
  \end{tabular}
\end{equation}

\subsubsection{Example: Multiplication of matrices.}
Given the two real-valued matrices $A(m\times p)$ and $B(p\times n)$ compute the matrix
product $C(m\times n)$, as defined by
\begin{equation}\label{mmultdef}
  C_{ij} = \sum_{k=1}^p A_{ik}*B_{kj}
\end{equation}
for $i=1,\dots,m$ and $j=1,\dots,n$.

The formulation of program (\ref{mmult}) follows from (\ref{mmultdef}) in a
straightforward manner.
\begin{Pascal}
var A:array[1..m,1..p]of real; B:array[1..p,1..n] of real; C:array[1..m,1..n] of real;
  i:1..m; j:1..n; k:1..p; s:real;
begin (*assignment of initial values to \pascal{A} and \pascal{B} *)
for i:=1to m do for j:=1 to n do
begin s:=0; for k:=1 to p do s:=s+A[i,k]*B[k,j]; C[i,j]:=s
end
end.
\end{Pascal}\pplabel{mmult}

\end{document}

```

Figure 2: Source code for the previous example

The Calculator Demo

Hans Hagen
pragma@pi.net

Abstract

Due to its open character, $\text{T}_{\text{E}}\text{X}$ can act as an authoring tool. This article demonstrates that by integrating $\text{T}_{\text{E}}\text{X}$, METAPOST, JavaScript and PDF, one can build pretty advanced documents. More and more documents will get the characteristics of programs, and $\text{T}_{\text{E}}\text{X}$ will be our main tool for producing them. The example described here can be produced with $\text{PDF}_{\text{T}_{\text{E}}\text{X}}$ as well as traditional $\text{T}_{\text{E}}\text{X}$.

Introduction

When Acrobat Forms were discussed at the $\text{PDF}_{\text{T}_{\text{E}}\text{X}}$ mailing list, Phillip Taylor confessed: “. . . they’re one of the nicest features of PDF”. Sebastian Ratz told us that he was “. . . convinced that people are waiting for forms.”. A few mails later he reported: “I just found I can embed JavaScript in forms, I can see the world is my oyster” after which in a personal mail he challenged me to pick up the Acrobat Forms plugin and wishing me “Happy JavaScripting”.

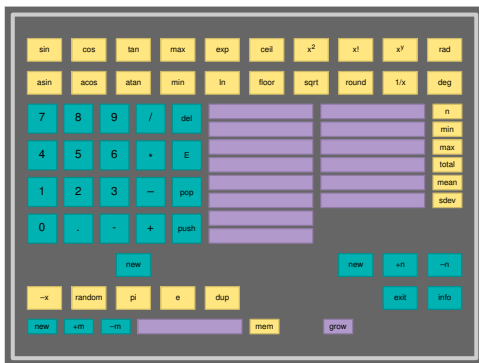


Figure 1 The calculator demo.

At the moment that these opinions were shared, I already had form support ready in CONTEXT , so picking up the challenge was a sort of natural behaviour. In this article I’ll describe some of the experiences I had when building a demo document that shows how forms and JavaScript can be used from within $\text{T}_{\text{E}}\text{X}$. I also take the opportunity to introduce some of the potentials of $\text{PDF}_{\text{T}_{\text{E}}\text{X}}$, so let’s start with introducing this extension to $\text{T}_{\text{E}}\text{X}$.

Where do we stand

While $\varepsilon\text{-T}_{\text{E}}\text{X}$ extends $\text{T}_{\text{E}}\text{X}$ ’s programming and typographic capabilities, $\text{PDF}_{\text{T}_{\text{E}}\text{X}}$ primarily acts at the back end of the $\text{T}_{\text{E}}\text{X}$ processor. Traditionally, $\text{T}_{\text{E}}\text{X}$

was (and is) used in the production chain:

$$\text{ASCII} \rightarrow \text{T}_{\text{E}}\text{X} \rightarrow \text{DVI} \rightarrow \text{whatever}$$

The most versatile process probably is:

$$\text{ASCII} \rightarrow \text{T}_{\text{E}}\text{X} \rightarrow \text{DVI} \rightarrow \text{POSTSCRIPT}$$

or even:

$$\text{ASCII} \rightarrow \text{T}_{\text{E}}\text{X} \rightarrow \text{DVI} \rightarrow \text{POSTSCRIPT} \rightarrow \text{PDF}$$

All functionality that $\text{T}_{\text{E}}\text{X}$ lacks, is to be taken care of by the DVI postprocessing program, and that’s why $\text{T}_{\text{E}}\text{X}$ can do color and graphic inclusion. Especially when producing huge files or files with huge graphics, the $\text{POSTSCRIPT} \rightarrow \text{PDF}$ steps can become a nuisance, if only in terms of time and disk space.

With PDF becoming more and more popular, it will be no surprise that Han The Thanh’s $\text{PDF}_{\text{T}_{\text{E}}\text{X}}$ becomes more and more popular too among the $\text{T}_{\text{E}}\text{X}$ users. With $\text{PDF}_{\text{T}_{\text{E}}\text{X}}$ we can reduce the chain to:

$$\text{ASCII} \rightarrow \text{T}_{\text{E}}\text{X} \rightarrow \text{PDF}$$

The lack of the postprocessing stage, forces $\text{PDF}_{\text{T}_{\text{E}}\text{X}}$ (i.e. $\text{T}_{\text{E}}\text{X}$) to take care of font inclusion, graphic inserts, color and more. One can imagine that this leads to lively discussions on the $\text{PDF}_{\text{T}_{\text{E}}\text{X}}$ mailing list and thereby puts an extra burden on the developer(s). Take only the fact that $\text{PDF}_{\text{T}_{\text{E}}\text{X}}$ is already used in real life situations while PDF is not stable yet.

To those who know PDF, it will be no surprise that $\text{PDF}_{\text{T}_{\text{E}}\text{X}}$ also supports all kind of hyper referencing. The version¹ I used when writing this article supports:

1. link annotations

¹ Currently I’m using β -version 1.12g.

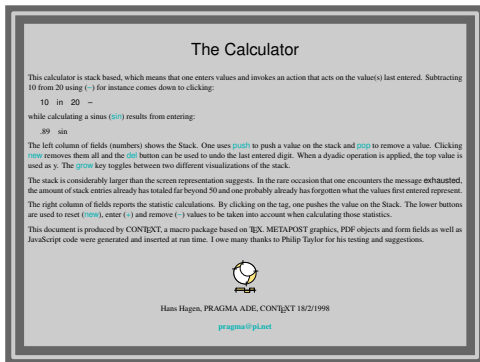


Figure 2 The help information screen.

2. screen handling
3. arbitrary annotations

where especially the last one is accompanied by:

4. form objects
5. direct objects

and of course there is also:

6. extensive font support

Be prepared: PDF_TE_X's font support probably goes (and certainly will go) beyond everything DVI drivers as well as Acrobat supports!

$T_{E}X$ stands in the typographic tradition and therefore has unsurpassed qualities. For many thousands of years people have trusted their ideas to paper and used glyphs for communication. The last decades however there has been a shift towards media like video, animations and interactive programs and currently these means of communication meet in hyper documents.

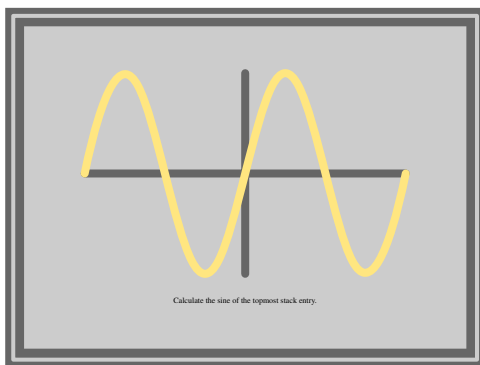


Figure 3 The $\sin(x)$ screen.

Now what has this to do with PDF_TE_X. Recently this program started to support the PDF annota-

tions other than the familiar hyperlink ones. As we will see later on, this enables users of $T_{E}X$ to enhance their documents with features that until now had to be programmed with dedicated tools, which could not even touch $T_{E}X$'s typographic quality. This means that currently $T_{E}X$ has become a tool for producing rather advanced documents within the typographic and (largely paper based) communication traditions. Even better, by using PDF as medium, one can produce very sophisticated interactive documents that are not bound to ill documented standards and programs and thereby stand a better chance to be accessible for future generations.

The calculator demo

The document described here is produced with CON_TE_XT. This document represents a full featured calculator which took me about two weeks to design and build. Most of the time was spend on defining METAPOST graphics that could explain the functionality of the buttons.² Extending CON_TE_XT for supporting JavaScript took me a few days and the rest of the time was spend on learning JavaScript itself.

The calculator demo was first developed using DVIPSONE and Acrobat. At that moment, PDF_TE_X did not yet provide the hooks needed, and the demo thereby served as a source of inspiration of what additional functionality was needed to let PDF_TE_X produce similar documents.

Throughout this article I show some of the screens that make up the calculator demo. These graphics are no screen dumps but just POSTSCRIPT inclusions. Just keep in mind that when using $T_{E}X$, one does not need bitmap screen dumps, but can use snapshots from the real document. A screen, although looking as one graphic, consist of a background with frame, a centered graphic, some additional text and an invisible active area the size of the gray center.

The demo implements a stack based calculator. The stack can optionally grow in two directions, depending on the taste of the user. Only the topmost entries of about 50 are visible.

The calculator demo, called `calculator.pdf`, itself can be fetched from the PDF_TE_X related site:

<http://www.tug.org/applications/pdftex>

or from the CON_TE_XT repository at:

² This included writing some auxiliary general purpose METAPOST macros.

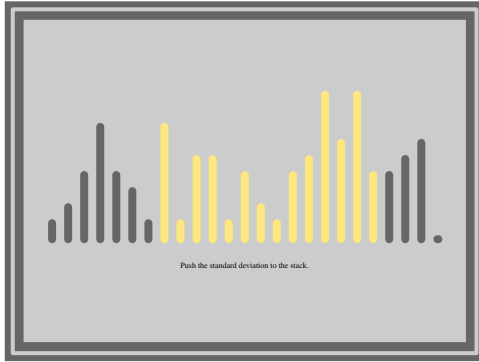


Figure 4 The standard deviation screen.

<http://www.ntg.nl/context>

The calculator is defined in one document source file, which not only holds the $\text{T}_{\text{E}}\text{X}$ code, but also contains the definitions of the METAPOST graphics and the JavaScript's. I considered including a movie (video) showing an animation of our company logo programmed in METAPOST and prepared in Adobe Premiere, but the mere fact that movies are (still) stored outside the PDF file made me remove this feature.

Now keep in mind that, when viewing the calculator PDF file, you're actually working with a document, not a program. A rather intelligent document for that matter, but still a document.

Forms and annotations

Before I go into details, I'll spend some words on forms and annotations in PDF. To start with the latter, annotations are elements in a PDF file that are not related to (typo)graphic issues, like movies and sound, hyper things, navigation and fill-in-forms. Formally annotations are dealt with by drivers plugged into the graphic engine, but in practice some annotations are handled by the viewer itself.

Forms in PDF are more or less the same as in HTML and once filled in can be send over the net to be processed. When filling in form fields, run time error checking on the input can prevent problems later on. Instead of building all kind of validation options into the form editor, such validations are handled by either a dedicated plugin, or better: by means of JavaScript. Therefore, one can attach such scripts to all kind of events related to form editing and one can launch scripts by associating them to active, that is clickable, areas on the screen.

So we've got fields, which can be used to let users provide input other than mere clicks on hyper

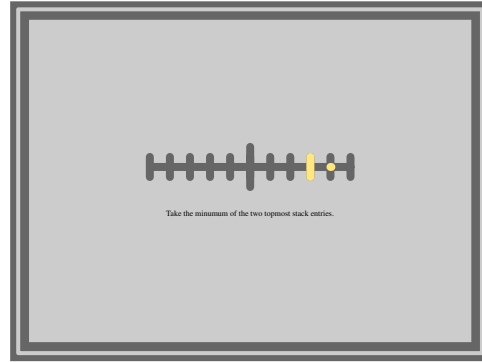


Figure 5 The $\min(x, y)$ screen.

links, we've got run time access to those fields using JavaScript, and we can let users launch such scripts by mouse events or keystrokes, either when entering data or by explicit request.

Currently entering data by using the keyboard is prohibited in the calculator. The main reason for this is that field allocation and access are yet sort of asynchronous and therefore lead to confusion.³

So, what actually happens in the calculator, is that a user clicks on a visualized key, thereby launching a JavaScript that in turn does something to field data (like adding a digit or calculating a sine), after which the field data is updated.

JavaScript

Writing this demo at least learned me that in fact support for JavaScript is just another sort of referencing and therefore needed incorporation in the general cross referencing scheme. The main reason is that for instance navigational tools like menus and buttons must have access to all cross reference mechanisms.

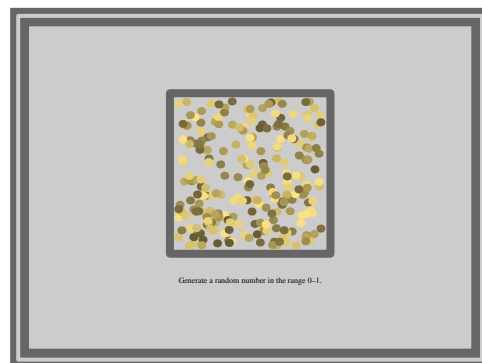


Figure 6 The random number screen.

³ Initializing a field from within JavaScript is not possible unless the viewer has (at some dubious moment) decided that the field indeed exists.

Consider for instance `\button{}`. We already support-
ed:

```
\button{...}[the chapter on whatever]
\button{...}[otherdoc::some topic]
\button{...}[previouspage]
\button{...}[PreviousJump]
```

Here the first reference is an internal one, often a chapter, a table or figure or a bibliography. The second one extends this class of references across documents. The third reference is a predefined internal one and the last reference gives access to viewer controls. As we can see: one scheme serves different purposes.

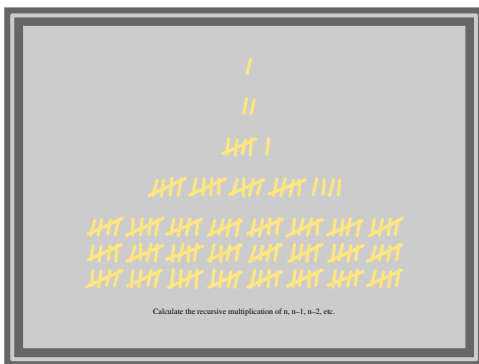


Figure 7 The period (.) screen.

Launching applications and following threads can quite easily be included in this scheme, but JavaScript support is different. In the calculator there are for instance 10 digit buttons that all do the same action and only differ in the digit involved. Here we want just one JavaScript to be reused 10 times. So instead of saying:

```
\button{0}[javascript 0]
\button{0}[javascript 1]
```

we want to express something like:

```
\def\SomeDigit#1%
  {\button{0}[javascript #1]}
```

```
\SomeDigit{4}
```

This means that in practice we need a referencing mechanism that:

- is able to recognize JavaScript
- is able to pass arguments to these scripts

So finally we end up with something:

```
\button{7}[JS(digit{7})]
```

This call tells the reference mechanism to access the JavaScript called `digit` and pass the value 7 to it. Actually defining the script comes down to just saying:

```
\startJScode{digit}
  Stack[Level] += String(JS_S_1);
  do_refresh(Level); //\ E
\stopJScode
```

One can pass as much arguments as needed. Here `JS_S_1` is the first string argument passed. Passing cross reference arguments is also possible. This enables us to let users jump to locations depending on their input. Such arguments are passed as `R{destination}` and can be accessed by `JS_R_1`.



Figure 8 The digit 7 screen.

In practice one will separate functions and calls by using preambles. Such preambles are document wide pieces of JavaScript, to be used whenever applicable.

```
\startJSpreamble{functions}
  // begin of common functions

  function do_digit(d)
  { Stack[Level] += String(d);
    do_refresh(Level) }

  // end of common functions //\ E
\stopJSpreamble
```

and:

```
\startJScode{digit}
  do_digit(JS_S_1); //\ E
\stopJScode
```

From these examples one can deduce that indeed the actual JavaScript code is included in the document source. It's up to \TeX to pass this information to

the PDF file, which in itself is not that trivial given the fact that one (1) has to strip comments, (2) has to convert some characters into legal PDF ones and (3) must pass arguments from $\text{T}_{\text{E}}\text{X}$ to JavaScript.

Simple cases like the digit code fragment, can also be passed as reference: $\text{JS}(\text{digit}\{1\})$. By default CONTEX T converts all functions present in the preambles into such references. One can organize JavaScripts into collections as well as postpone inclusion of preambles until they are actually used.

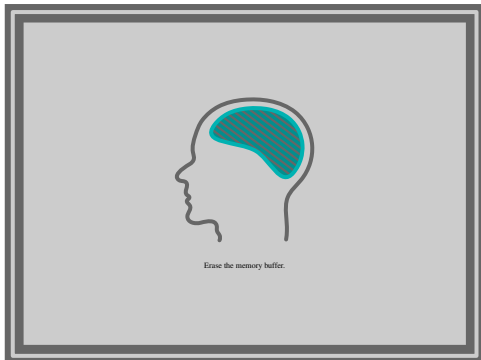


Figure 9 The memory erase screen.

Currently the only problem with including preambles lays in the mere fact that Acrobat pdfmarks⁴ do not yet offer a mechanism to enter the JavaScript entries in the appropriate place in the document catalog, without spoiling the collected list of named destinations. Because CONTEX T can be instructed to use page destinations when possible, I could work around this (temporary) Acrobat pdfmark and $\text{PDF}_{\text{T}_{\text{E}}\text{X}}$ limitation. At the time this article is published, $\text{PDF}_{\text{T}_{\text{E}}\text{X}}$ probably handles this conceptual weak part of PDF in an adequate way.

METAPOST graphics

All graphics are generated at run time using METAPOST. Like the previous mentioned script, METAPOST code is included in the source of the document. For instance, the graphic representing π is defined as:

```
\startuseMPgraphic{pi}
  pickup pencircle scaled 10;
  draw fullcircle
    scaled 150
    withcolor .4white;
  linecap := butt;
  ahlength := 25;
  drawarrow halfcircle
    scaled 150
```

```
withcolor \MPcolor{action};
\stopuseMPgraphic
and called
\useMPgraphic{pi}
```

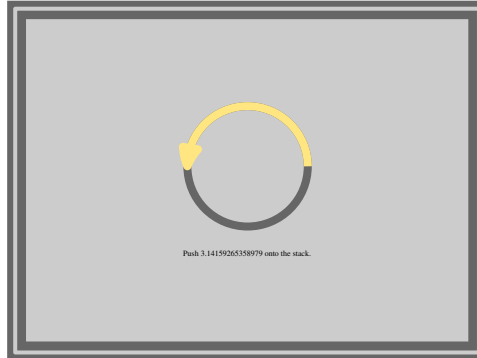


Figure 10 The π screen.

Just like the JavaScript preamble we can separate common METAPOST functions by defining inclusions. The next one automatically loads a module with some auxiliary macros.

```
\startMPinclusions
  input mp-tool;
\stopMPinclusions
```

The mechanism for including METAPOST graphics is also able to deal with reusing graphics and running METAPOST itself from within $\text{T}_{\text{E}}\text{X}$. In CONTEX T all processed METAPOST graphics are automatically translated into PDF by $\text{T}_{\text{E}}\text{X}$ itself, colors are converted to the current color space, and text is dealt with accordingly. Of course one needs to take care of proper tagging, but the next macro does this well:

```
\def\SomeShape#1#2%
  {\startreuseMPgraphic{shape:#1#2}
   draw fullcircle
     xscaled #1
     yscaled #2
   \stopreuseMPgraphic
   \reuseMPgraphic{shape:#1#2}}
```

Now we can say:

```
\SomeShape{100pt}{200pt}
\SomeShape{150pt}{180pt}
\SomeShape{120pt}{110pt}
```

Which just inserts three graphics with different sizes but similar line widths.

⁴ These are extensions to the POSTSCRIPT language.

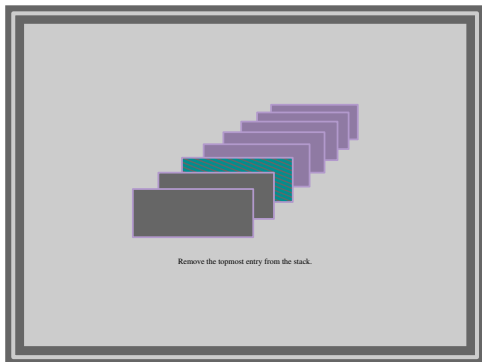


Figure 11 The pop stack screen.

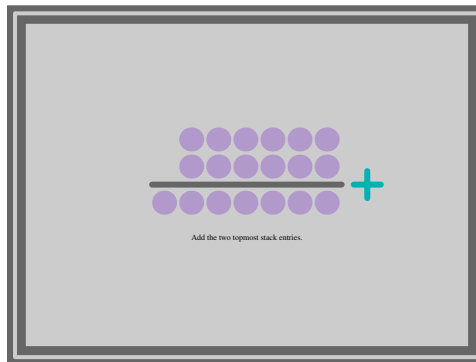


Figure 12 The addition (+) screen.

Backgrounds

Now how do we attach such shapes to the buttons? Here we introduce a feature common to all framed things in `CONTEXT`, called overlays. Such an overlay is defined as:

```
\defineoverlay
  [shape]
  [\MPshape
    {\overlaywidth}
    {\overlayheight}
    {\overlaycolor}]
```

The shape called `\MPshape` is defined as:

```
\def\MPshape#1#2#3%
  {\startreusableMPgraphic{fs:#1#2#3}
    path p ;
    p := unitsquare
      xscaled #1
      yscaled #2;
    color c ;
    c := #3 ;
    fill p
      withcolor c ;
    draw p
      withpen pencircle scaled 1.5
      withcolor .8c ;
    \stopreusableMPgraphic
    \reuseMPgraphic{fs:#1#2#3}}
```

Such an overlay is bound to a particular framed thing by saying:

```
\setupbuttons[background=shape]
```

Here the right dimensions are automatically passed on to the overlay mechanism which in turn invokes `METAPOST`.

The calculator demo proved me that it is rather useful to have stacked backgrounds. Therefore the buttons, which have both a background (the

`METAPOST` drawn shape) and behind that a sort of help button that is activated by clicking on the surroundings of the button, have their backgrounds defined as:

```
\setupbuttons
  [background={infobutton,shape}]
```

Actually we're stacking from back to top: an info button, the key bound button, the background graphic and the text. One rather tricky side effect is that stacked buttons interfere with the way active areas are output. In this particular case we have to revert the order of the active areas by saying `\reversegototruer`.

Object reuse

The button and background graphics are generated once and used more than once. We already mentioned that `METAPOST` graphics can be reused. In practice this comes down to producing the graphic once and including it many times. In PDF however, one can also include the graphic once and refer to it many times. In PDF such reused objects are called forms, a rather unfortunate naming. So, in the calculator demo, all buttons with common shapes as well as the backgrounds are included only once. One can imagine that extending `TEX` with such features leads to interesting discussions on the `PDFTEX` discussion list.

Forms

Although still under construction, `CONTEXT` supports PDF fill-in-forms. The calculator demo demonstrates that such forms can be used as a (two way) communication channel to the user. Stack values, statistics and memory content are stored and presented in form fields, defined by saying something like:

Visual Debugging in T_EX

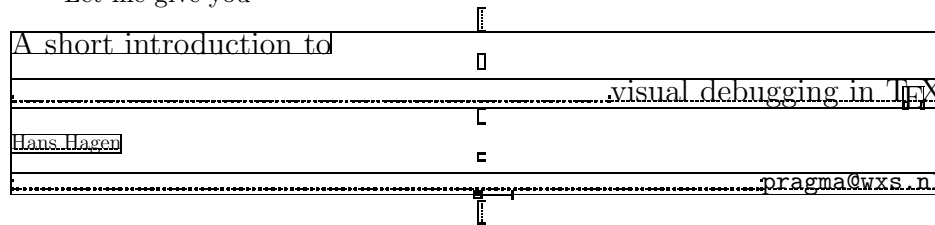
Part 1: The Story

Hans Hagen
PRAGMA ADE
Ridderstraat 17
8061GH Hasselt NL
pragma@wxs.nl

Abstract

This article shows some features of the visual debugging module I wrote in 1996 as part of the CONTEX_T macro package. This module visualizes the low level typesetting components, like boxes, skips, glues and fills. Although beyond the scope of this article, they also let surface some behavior that often goes unnoticed.

Let me give you



This kind of fancy heading shows some dotted lines, rules and peculiar visual symbols. A closer observation shows that in fact it is some endoscopic view into what is often called T_EX's stomach. For those readers who have planned to skip the rest of this article, here is how the magic is done:

```
\input supp-vis \showmakeup
```

For those who want to take a closer look at all those kerns, skips and penalties, this article can be of some help. Although this kind of stuff often attracts the more hacking type of reader, the module described here can be of great help and provide a lot of fun to all T_EX users, whatever macro package they use.

When T_EX builds paragraphs and pages, it takes a lot into account. Even after years of writing macros the interference of skips, kerns, penalties, boxes and rules sometimes surprises me. One must always be aware of interline skips, top of page skips, good breaks and no breaks, either user supplied or system generated.

The idea to build some visualization macros was born while I was documenting the source of CONTEX_T. Because this package is quite complete, the full documentation will be laid down in thousands of pages. Such technical documentation cannot go without showing how things are done. Because most macros at the user level have some visual impact, I decided to build a visualization tool. After having written this bunch of macros, their second purpose soon became visual debugging.

The concept is rather simple: replace the primitives `\.box`, `\.skip`, `\kern`, `\penalty`, `\.glue`, `\.ss`, `\.fil`. and `\.fil.neg` by macros that makes them visible. Most advanced T_EX tutorials give examples of adapting the primitive `\par`, but somehow tampering with other T_EX primitives is considered more tricky. Although the name primitive suggests that they are somehow fixed, even primitives can be `\let`'d or `\def`'d to something else. Temporarily superseding the `\font` primitive is for instance needed when one wants to postpone loading of fonts in Plain T_EX.

One can imagine that replacing `\hbox` with something else can have disastrous consequences. Primitives like `\setbox` expect a box and setting `\hbox` to `\relax` will surely lead to loud complaints. Some first experiments showed however that substitution was surprisingly easy. More time was spent on finding a sort of replacement that does not conflict visually when more primitives are given in a row.

Let's start with a well known piece of text. We've blown it up a bit, so we can see what happens.

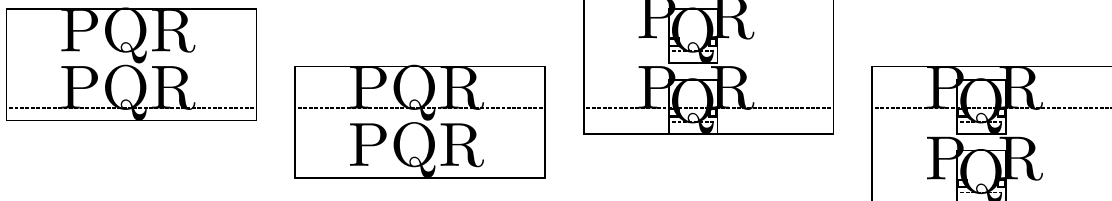


Here we see a T, followed by a kern, a boxed E, another kern and an X. The kerns have a negative sign and are visualized as small rectangles. Negative values are drawn left of their insertion point. The second T_EX has exaggerated cues.

The three uppercase characters that make up T_EX have no descenders. The next example shows a few more T_EXed characters. This time we've got them boxed, so we can see what happens to the baseline of this combination of characters. Lowering the B and Q does not influence the baseline, which is what we expect.



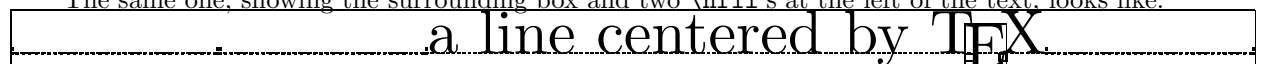
Vertical boxes come in two flavors. The default vertical box `\vbox` inherits its baseline from the last line, while `\vtop` takes the baseline of the first line.



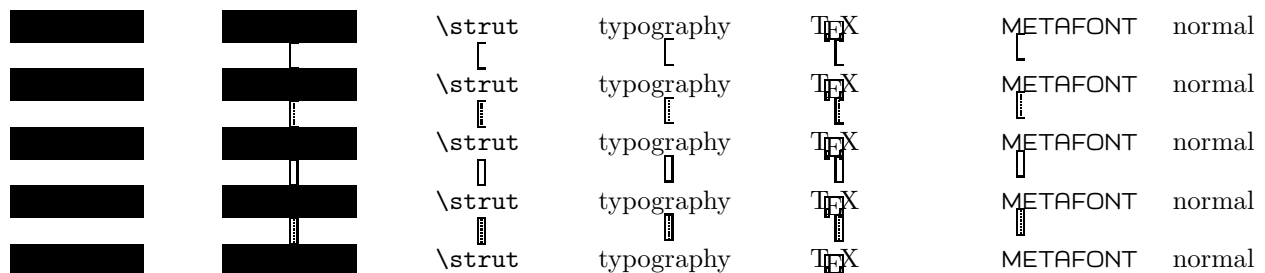
Visualization of fills is no problem either. In the centered line shown below the piece of text has some `\hfil`'s around it.



The same one, showing the surrounding box and two `\hfil`'s at the left of the text, looks like:



When using substitutes for the primitives mentioned, keeping the spacing intact is not always trivial. Especially the vertical spacing is very sensitive to interference. The next examples show us that at least normal situations can be handled well.

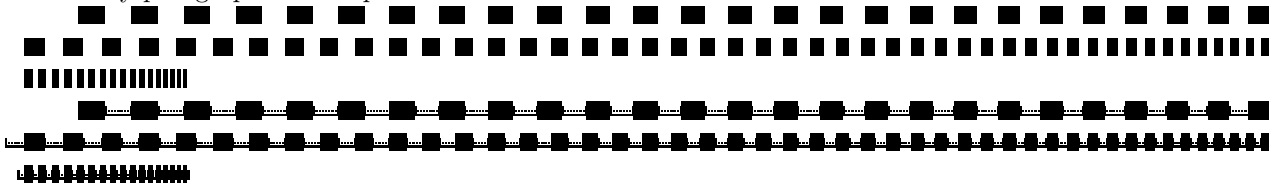


Here we see some positive vertical cues. Their negative counterparts are drawn left of the axis. Top down we see a skip, another skip with some stretch, a kern and some glue. A penalty of 100 looks like this and can be negative too. Skips, kerns and glue, which by the way is a Plain T_EX macro and not a primitive, are shown at their natural size. Penalties are drawn in ranges, which are tuned to the most common cases. Combinations of penalties show up all right as we can see in where we have inserted penalties of 10000, 100 and 1.

Horizontal spacing is less sensitive than vertical spacing. Here we don't have to take interline spacing and previous depths into account. Just to prove that things work, we show a similar example here. As a bonus we've added `\hss`.

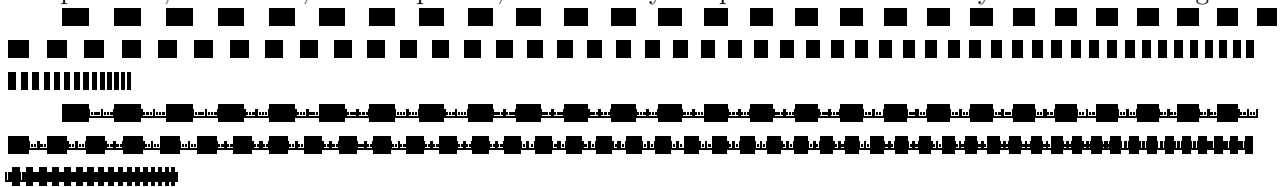


When we are typesetting in horizontal mode we have to preserve linebreaking. The next example shows a dummy paragraph with skips.



In this example it's hard to see that the stretch is equally distributed around the skip. The next line of text shows this feature in full glory. This feature is disabled by default.
hello big big world

Now look what happens when we combine two horizontal skips. This time TeX is not able to remove the visual cues. A similar situation occurs at a pagebreak. This kind of tricky situation can only be solved by an invisible kind of box, which is unfortunately not part of TeX. Of course we can backtrack skips, kerns and penalties, but such a, still not perfect, solution only complicates the macros beyond understanding.



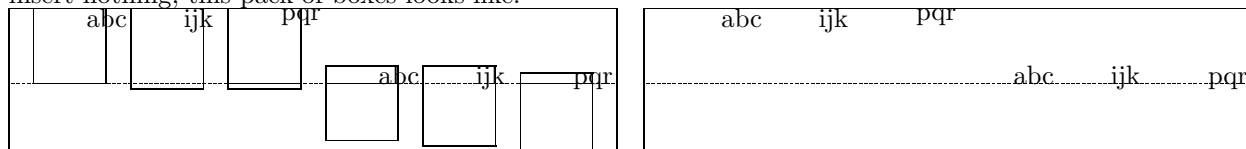
Mathematical spacing is implemented too, but due to the font-bound character, its visualization is the least impressive: $x\lrcorner y$ and $x\llcorner y$ for math kern and math skip of 7 mu.

The next set of examples shows how vertical boxes are aligned when pasted together in a horizontal box. When I was messing around a bit with these samples, I became aware of some side effects that normally go unnoticed probably because they are quite natural. Confronted with these effects, I first thought that the visualization macros were somehow responsible, but additional testing proved otherwise. Of course one can never be sure, but rereading some paragraphs in Victor Eijkhout's "TeX by Topic" taught me that indeed such effects occur.

The samples are built up in the following way. Here the dots stand for some trailing text and/or macros.

```
\hbox to \hsize
{\hss
 \hsize.15\hsize
 \vbox to 1cm{abc\par ...}\hss
 \vbox to 1cm{ijk\par ...}\hss
 \vbox to 1cm{pqr\par ...}\hss
 \vtop to 1cm{abc\par ...}\hss
 \vtop to 1cm{ijk\par ...}\hss
 \vtop to 1cm{pqr\par ...}\hss}
```

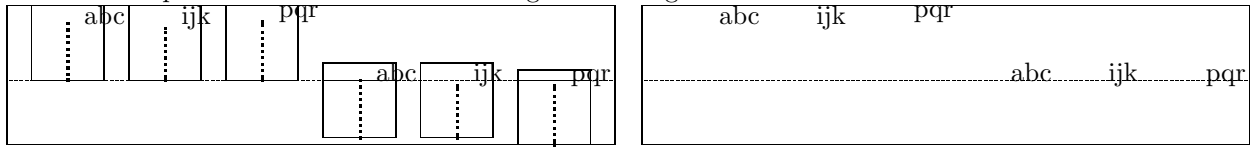
We show both the visualized example and the natural one. The latter illustrates compatibility. When we insert nothing, this pack of boxes looks like:



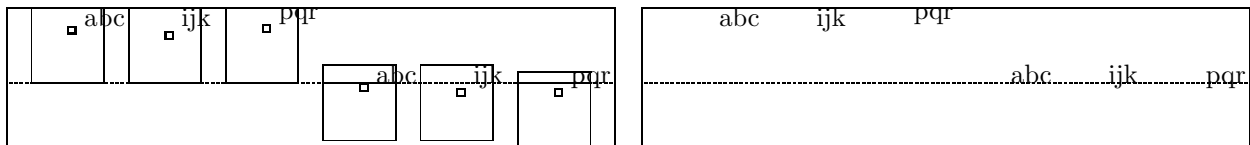
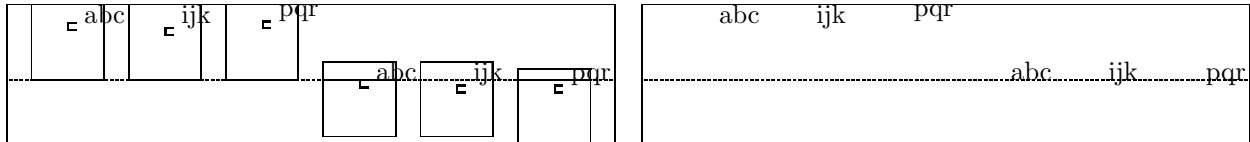
The first box has the height we expect. The second and third box also have the desired height, but here the depth of the j and q has migrated to the surrounding box. The height and depth of the fourth box totals to 1 cm, and we don't recognize the 1 cm in one of those dimensions. The last two boxes behave a bit unexpected. Here the depth is added to the height we specified. These last three situations teach us that

specifying the height of a `\vtop` does not always make that much sense.

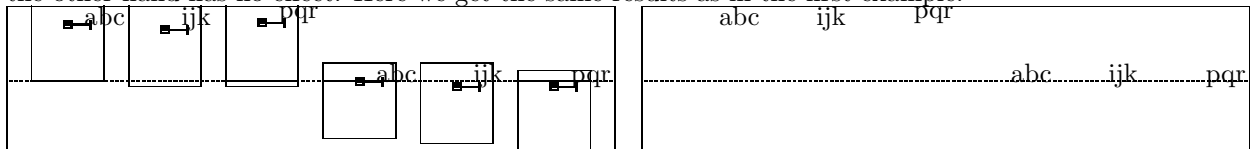
Now watch what happens when we add a `\vss`. This time the `ijk` and `pqr` boxes behave as expected and we end up with six boxes of 1 cm. Seeing is believing.



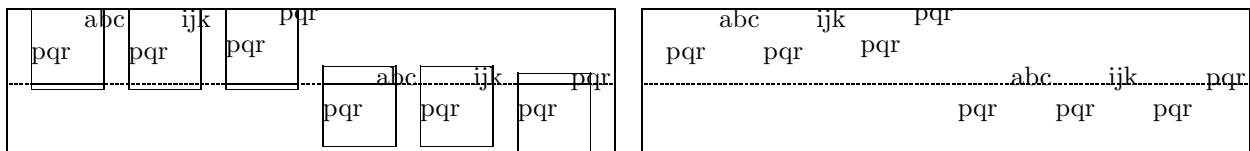
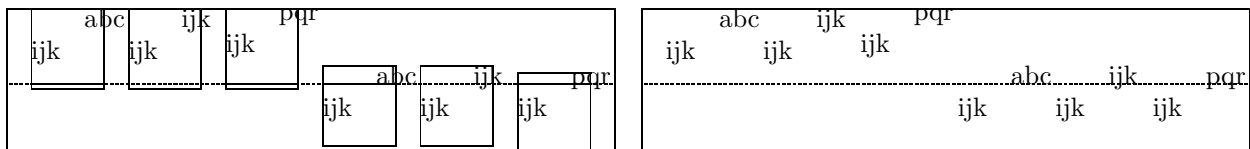
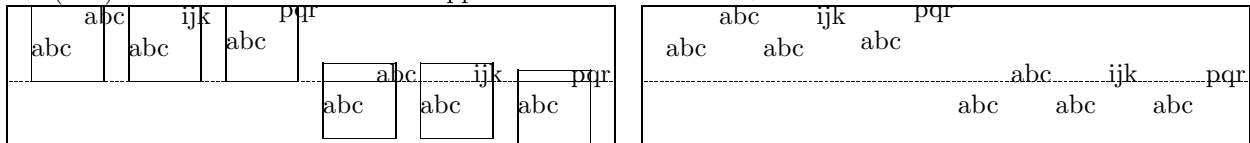
In most cases, one will add some kind of glue to a box, just to get rid of those underfull messages. It's good to be aware of the fact that adding glue does a bit more. Adding a `\vskip` or `\kern` has the same effect.



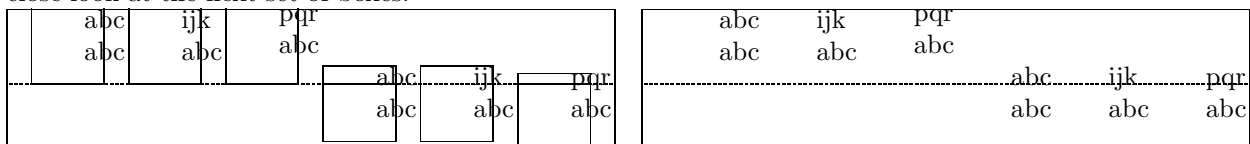
Adding a very large skip or kern makes no difference so we stick to these 3 pt examples. A penalty on the other hand has no effect. Here we get the same results as in the first example.



When we add some boxed text, the height and depth of the surrounding box depend on the depth of the (last) line. Here we show what happens when we insert an `\hbox`.



When we put the characters in an `\hbox` and `\unhbox` this box, we get different results. Just take a close look at the next set of boxes.



abc ijk	ijk ijk	pqr ijk	abc ijk	ijk ijk	pqr ijk
abc pqr	ijk pqr	pqr pqr	abc pqr	ijk pqr	pqr pqr

abc	ijk	pqr	abc	ijk	pqr
ijk	ijk	ijk	ijk	ijk	ijk

Things looks different when we add a `\vbox`. Just adding one looks like this:

abc abc	ijk abc	pqr abc	abc abc	ijk abc	pqr abc
abc ijk	ijk ijk	pqr ijk	abc ijk	ijk ijk	pqr ijk
abc pqr	ijk pqr	pqr pqr	abc pqr	ijk pqr	pqr pqr

abc	ijk	pqr	abc	ijk	pqr
abc	abc	abc	abc	ijk	pqr
ijk	ijk	ijk	ijk	ijk	ijk
pqr	pqr	pqr	pqr	pqr	pqr

Adding an `\unvbox`'ed one looks a bit different. This kind of test can be both very confusing and instructive. It's a challenge to deduce some systematic behavior from them.

abc abc	ijk abc	pqr abc	abc abc	ijk abc	pqr abc
abc ijk	ijk ijk	pqr ijk	abc ijk	ijk ijk	pqr ijk
abc pqr	ijk pqr	pqr pqr	abc pqr	ijk pqr	pqr pqr

abc	ijk	pqr	abc	ijk	pqr
abc	abc	abc	abc	ijk	pqr
ijk	ijk	ijk	ijk	ijk	ijk
pqr	pqr	pqr	pqr	pqr	pqr

I could show some more examples, like vertical boxes with more lines or `\vtop`'s. The examples shown here at least make clear that when we start manipulating boxes, we have to be aware of side effects.

Close reading of the `TEXbook` teaches that the effects of the skip stretch components `plus` and `minus` sometimes depend on the context. Take a look at the set of boxes in table 1.

Line- and pagebreaks can in no way be handled 100% perfectly. `TEX` clears out redundant skips and penalties when crossing lines and pages. Making skips and penalties visible calls for the use of boxes and rules. A more perfect visualizer can be built when two more box primitives are made available: `\hnop` and `\vnop`. Both primitives should act like normal boxes when being manipulated, but should be kept out of paragraph and pagebreak calculation. They should be visible in the output but invisible for `TEX` itself. Lacking these primitives, visualization of sequences of skips and penalties will lead to non-compatible results.

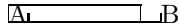
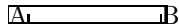
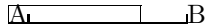
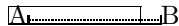
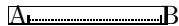
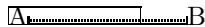




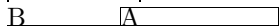
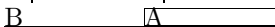

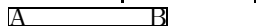


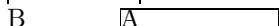

<code>\hbox to 5em {A\hskip 5em B}</code>	
<code>\hbox to 6em {A\hskip 5em B}</code>	
<code>\hbox to 5em {A\hskip 6em B}</code>	
<code>\hbox to 5em {A\hskip 5em plus 1em B}</code>	
<code>\hbox to 6em {A\hskip 5em plus 1em B}</code>	
<code>\hbox to 5em {A\hskip 6em plus 1em B}</code>	
<code>\hbox to 5em {A\hskip 5em minus 1em B}</code>	
<code>\hbox to 6em {A\hskip 5em minus 1em B}</code>	
<code>\hbox to 5em {A\hskip 6em minus 1em B}</code>	
<code>\hbox to 5em {A\hskip -5em B}</code>	
<code>\hbox to 6em {A\hskip -5em B}</code>	
<code>\hbox to 5em {A\hskip -6em B}</code>	
<code>\hbox to 5em {A\hskip -5em plus 1em B}</code>	
<code>\hbox to 6em {A\hskip -5em plus 1em B}</code>	
<code>\hbox to 5em {A\hskip -6em plus 1em B}</code>	
<code>\hbox to 5em {A\hskip -5em minus 1em B}</code>	
<code>\hbox to 6em {A\hskip -5em minus 1em B}</code>	
<code>\hbox to 5em {A\hskip -6em minus 1em B}</code>	

Table 1

Like the colored verbatim modules described in a previous article, the visual debugger module can be used on top of Plain T_EX. Both modules only use a few general system macros, which are supplied in a small miscellaneous module. For CONTEX_T users, visualization is always available, because it's just one of the standard features. For users of Plain T_EX (or for those who use other packages) the next commands will do the trick:

```
\input supp-vis
```

When this module is loaded, `\showmakeup` will turn on the visualization. Users can turn on and off some features, like alignment of vertical cues, individual categories of cues and the visible baseline. The macros and features are explained in detail in the documented module itself.

The `supp` stands for general support. The symmetrical verbatim module, which supports typesetting of colored T_EX sources that we presented in a previous article, belongs to this category too. When used outside CONTEX_T, both modules automatically fall back on a small module `supp-mis`, which implements poor mans alternatives for a few system macros.

Visualization can best be used grouped. Depending on the number of primitives used, the output can be huge when one processes whole pages. Plain T_EX's pagebody routine is both simple and effective. Unfortunately, the more flexibility one wants, the more complicated this routine becomes. In CONTEX_T for instance this routine has to deal with multiple headers and footers, backgrounds, logos, multiple margins, interaction menus, navigational tools and a few more. Therefore we turn off visualization as long as we are building the page. The same goes for multi-column handling and some Plain T_EX macros like `\llap` and `\rlap`.

In Plain T_EX it's not that hard to turn things off temporarily. Just give the next code a try:

```
\input supp-vis
\output
  {\dontshowcomposition\plainoutput}
\showmakeup
\hbox{so much}
\ejct
\hbox{for now}
\end
```

In CONTEX_T there are some more similar facilities, like general layout, `\strut` and baseline visualization. At the moment, the functionality of this module is limited to the primitives mentioned. We already visualize

the mathematical skips, but when needed, we will extend this module with some useful math debugging facilities. A year from now, this module probably will be a bit more advanced anyway.

I could show some more instructive examples, but for producing those, I have to depend a bit too much on CONTEXT for processing. For the same reason the next article, which describes the module itself, lacks some useful functionality.

Let's summarize the cues. Positive horizontal cues are drawn on top of and negative ones under the baseline. The negative cues are drawn in the negative direction. Vertical cues are drawn left or right of the current point (or halfway the `\hsize`) and they too honor the direction. In the table 2 next table we only show the horizontal cues.

<code>\hss</code>	A.....B	
<code>\hfil</code>	A.....B	A B
<code>\hfill</code>	A.....B	A B
<code>\hskip 5em</code>	A_____B	B_____A
<code>\hskip 5em plus 1em</code>	A_____B	B_____A
<code>\kern 5em</code>	A=====B	B=====A
<code>\hglue 5em plus 1em</code>	A=====B	B=====A
<code>\penalty 200</code>	A B	A B
<code>\mskip 50mu plus 1mu</code>	A _____B	B_____A
<code>\mkern 50mu</code>	A _____B	B_____A

Table 2

Kerns and penalties are treated according to the current mode, which is horizontal or vertical. Zero cues are a special case. A zero horizontal skip for instance shows up as | , a kern looks like | and a zero penalty becomes | . As far as possible, different kinds of cues add up nicely.

ConTEXt

Visualization

category: ConTEXt Support Macros

version: 1996.10.21

date: December 18, 1998

author: Hans Hagen

copyright: PRAGMA / Hans Hagen & Ton Otten

Although an integral part of ConTeXt, this module is one of the support modules. Its stand alone character permits use in PLAIN T_EX or T_EX based macropackages.

Depending on my personal needs and those of whoever uses it, the macros will be improved in terms of visualization, efficiency and compatibility. These rather low level visualization macros are supplemented by ones that can visualize baselines, the page layout and whatever deserves attention. Most of those macros can be found in `core-vis` and other core modules. Their integration in ConTeXt prohibits generic applications.

```
1 \ifx \undefined \writestatus \input supp-mis.tex \fi
```

One of the strong points of T_EX is abstraction of textual input. When macros are defined well and do what we want them to do, we will seldom need the tools present in What You See Is What You Get systems. For instance, when entering text we don't need rulers, because no manual shifting and/or alignment of text is needed. On the other hand, when we are designing macros or specifying layout elements, some insight in T_EX's advanced spacing, kerning, filling, boxing and punishment abilities will be handy. That's why we've implemented a mechanism that shows some of the inner secrets of T_EX.

```
2 \writestatus{loading}{Context Support Macros / Visualization}
```

In this module we are going to redefine some T_EX primitives and PLAIN macro's. Their original meaning is saved in macros with corresponding names, preceded by `normal`. These original macros are (1) used to temporarily restore the old values when needed and (2) used to prevent recursive calls in the macros that replace them.

```
3 \unprotect
```

```
4 \let\visualvrule\vrule
\let\visualhrule\hrule
```

`\dontinter..` Indentation, left and/or right skips, redefinition of `\par` and assignments to `\everypar` can lead to unwanted results. We can therefore turn all those things off with `\dontinterfere`.

```
5 \def\dontinterfere%
  {\everypar = {}}%
  \let\par = \endgraf
  \parindent = \!!zeropoint
  \parskip = \!!zeropoint
  \leftskip = \!!zeropoint
  \rightskip = \!!zeropoint
  \relax}
```

`\dontcompl..` In this module we do a lot of box manipulations. Because we don't want to be confronted with too many over- and underfull messages we introduce `\dontcomplain`.

```
6 \def\dontcomplain%
  {\hbadness = \!!tenthousand
  \hfuzz = \maxdimen
  \vbadness = \!!tenthousand
  \vfuzz = \maxdimen}
```

`\normaloff..` The next hack is needed because in ConTeXt we redefine `\offinterlineskip`.

```

7 \ifx\undefined\normaloffinterlineskip
  \let\normaloffinterlineskip=\offinterlineskip
\fi

```

`\normalhbox`
`\normalvbox`
`\normalvtop` There are three types of boxes, one horizontal and two vertical in nature. As we will see later on, all three types are to be handled according to their orientation and baseline behavior. Especially `\vtop`'s need our special attention.

```

8 \let\normalhbox = \hbox
  \let\normalvbox = \vbox
  \let\normalvtop = \vtop
  \let\normalvcenter = \vcenter

```

`\normalhskip`
`\normalvskip` Next come the flexible skips, which come in two flavors too. Like boxes these are handled with \TeX primitives.

```

9 \let\normalhskip = \hskip
  \let\normalvskip = \vskip

```

`\normalpen..`
`\normalkern` Both penalties and kerns are taken care of by mode sensitive primitives. This means that when making them visible, we have to take the current mode into account.

```

10 \let\normalpenalty = \penalty
    \let\normalkern = \kern

```

`\normalhglue`
`\normalvglue` Glues on the other hand are macro's defined in PLAIN \TeX . As we will see, their definitions make the implementation of their visible counterparts a bit more \TeX nical.

```

11 \let\normalhglue = \hglue
    \let\normalvglue = \vglue

```

`\normalmkern`
`\normalmskip` Math mode has its own spacing primitives, preceded by `m`. Due to the relation with the current font and the way math is typeset, their unit μ is not compatible with other dimensions. As a result, the visual appearance of these primitives is kept primitive too.

```

12 \let\normalmkern = \mkern
    \let\normalmskip = \mskip

```

`\hfilneg`
`\vfilneg` Fills can be made visible quite easy. We only need some additional negation macros. Because PLAIN \TeX only offers `\hfilneg` and `\vfilneg`, we define our own alternative double ll'ed ones.

```

13 \def\hfillneg%
    {\normalhskip\!!zeropoint \!!plus-1fill\relax}

```

```

14 \def\vfillneg%
    {\normalvskip\!!zeropoint \!!plus-1fill\relax}

```

`\normalhss`
`\normalhfil`
`\normalhfill`
`\normalvss`
`\normalvfil`
`\normalvfill` The positive stretch primitives are used independant and in combination with `\leaders`.

```

15 \let\normalhss = \hss
    \let\normalhfil = \hfil
    \let\normalhfill = \hfill
    \let\normalvss = \vss
    \let\normalvfil = \vfil
    \let\normalvfill = \vfill

```

`\normalhfi..`
`\normalhfi..`
`\normalvfi..`
`\normalvfi..` Keep in mind that both `\hfillneg` and `\vfillneg` are not part of PLAIN \TeX and therefore not documented in standard \TeX documentation. They can nevertheless be used at will.

```

16 \let\normalhfilneg = \hfilneg
    \let\normalhfillneg = \hfillneg
    \let\normalvfilneg = \vfilneg
    \let\normalvfillneg = \vfillneg

```

Visualization is not always wanted. Instead of turning this option off in those (unpredictable) situations, we just redefine a few PLAIN macros.

```

17 \def\rlap#1{\normalhbox to \!!zeropoint{#1\normalhss}}
    \def\llap#1{\normalhbox to \!!zeropoint{\normalhss#1}}
18 \def~{\normalpenalty\!!tenthousand\ }

```

`\makeruled..` Ruled boxes can be typeset in many ways. Here we present just one alternative. This implementation may be a little complicated, but it supports all three kind of boxes. The next command expects a `<box>` specification, like:

```
\makeruledbox0
```

`\baseliner..` We can make the baseline of a box visible, both dashed and as a rule. The line is drawn on top of the
`\baselinef..` baseline. All we have to say is:

```
\baselineruletrue
\baselinefilltrue
```

At the cost of some overhead these alternatives are implemented using `\if`'s:

```

19 \newif\ifbaselinerule \baselineruletrue
    \newif\ifbaselinefill \baselinefillfalse

```

`\iftoprule` Rules can be turned on and off, but by default we have:

```

\ifbottomr.. \topruletrue
\ifleftrule  \bottomruletrue
\ifrightrule \leftruletrue
              \rightruletrue

```

As we see below:

```

20 \newif\iftoprule \topruletrue
    \newif\ifbottomrule \bottomruletrue
    \newif\ifleftrule \leftruletrue
    \newif\ifrightrule \rightruletrue

```

`\boxrulewi..` The width in the surrounding rules can be specified by assigning an appropriate value to the dimension used. This module defaults the width to:

```
\boxrulewidth=.2pt
```

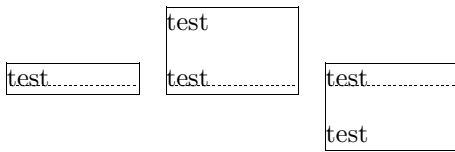
Although we are already low on `<dimensions>` it's best to spend one here, mainly because it enables easy manipulation, like multiplication by a given factor.

```

21 \newdimen\boxrulewidth \boxrulewidth=.2pt

```

The core macro `\makeruledbox` looks a bit hefty. The manipulation at the end is needed because we want to preserve both the mode and the baseline. This means that `\vtop`'s and `\vbox`'es behave the way we expect them to do.



The `\cleaders` part of the macro is responsible for the visual baseline. The `\normalhfill` belongs to this primitive too. By storing and restoring the height and depth of box #1, we preserve the mode.

```

22 \def\makeruledbox#1%
    {\edef\ruledheight{\the\ht#1}%
     \edef\ruleddepth {\the\dp#1}%
     \edef\ruledwidth {\the\wd#1}%
     \setbox\scratchbox=\normalvbox
     {\dontcomplain
      \normaloffinterlineskip
      \visualhrule
       \!!height\boxrulewidth
       \iftoprule\else\!!width\!!zeropoint\fi
      \normalvskip-\boxrulewidth
      \normalhbox to \ruledwidth
      {\visualvrule
       \!!height\ruledheight
       \!!depth\ruleddepth
       \!!width\iflefttrule\else0\fi\boxrulewidth
       \ifdim\ruledheight>\!!zeropoint \else \baselinerulefalse \fi
       \ifdim\ruleddepth>\!!zeropoint \else \baselinerulefalse \fi
       \ifbaselinerule
        \ifdim\ruledwidth<20\boxrulewidth
         \baselinefilltrue
        \fi
        \cleaders
         \ifbaselinefill
          \visualhrule
           \!!height\boxrulewidth
         \else
          \normalhbox
           {\normalhskip2.5\boxrulewidth
            \visualvrule
             \!!height\boxrulewidth
             \!!width5\boxrulewidth
            \normalhskip2.5\boxrulewidth}%
          \fi
        \fi
        \normalhfill
        \visualvrule
         \!!width\ifrighttrule\else0\fi\boxrulewidth}%
      \normalvskip-\boxrulewidth
      \visualhrule
       \!!height\boxrulewidth
       \ifbottomrule\else\!!width\!!zeropoint\fi}%
     \wd#1=\!!zeropoint
     \setbox#1=\ifhbox#1\normalhbox\else\normalvbox\fi
     {\normalhbox{\box#1\lower\ruleddepth\box\scratchbox}}%
  
```

```
\ht#1=\ruledheight
\wd#1=\ruledwidth
\dp#1=\ruleddepth}
```

Just in case one didn't notice: the rules are in fact layed over the box. This way the contents of a box cannot visually interfere with the rules around (upon) it. A more advanced version of ruled boxes can be found in one of the core modules of ConTeXt. There we take offsets, color, rounded corners, backgrounds and alignment into account too.

```
\ruledhbox
\ruledvbox
\ruledvtop
\ruledvcen..
```

These macro's can be used instead of `\hbox`, `\vbox`, `\vtop` and, when in math mode, `\vcenter`. They just do what their names state. Using an auxiliary macro would save us a few words of memory, but it would make their appearance even more obscure.

one.two.three.four.five

```
\hbox
{\strut
one
two
\hbox{three}
four
five}
```

```
23 \unexpanded\def\ruledhbox%
    {\normalhbox\bgroup
     \dowithnextbox{\makeruledbox\nextbox\box\nextbox\egroup}%
     \normalhbox}
```

first line
second line
third line
fourth line
fifth line.....

```
\vbox
{\strut
first line \par
second line \par
third line \par
fourth line \par
fifth line
\strut }
```

```
24 \unexpanded\def\ruledvbox%
    {\normalvbox\bgroup
     \dowithnextbox{\makeruledbox\nextbox\box\nextbox\egroup}%
     \normalvbox}
```

first line.....
second line
third line
fourth line
fifth line

```
\vtop
{\strut
first line \par
second line \par
third line \par
fourth line \par
fifth line
\strut }
```



```

25 \unexpanded\def\ruledvtop%
    {\normalvtop\bgroup
     \dowithnextbox{\makeruledbox\nextbox\box\nextbox\egroup}%
     \normalvtop}

```

	alfa beta gamma	
alfa beta		alfa beta

```

\hbox
  {${\vcenter{\hsize.2\hsize
    alfa \par beta}$
   ${\vcenter to 3cm{\hsize.2\hsize
    alfa \par beta \par gamma}$
   ${\vcenter{\hsize.2\hsize
    alfa \par beta}$}

```

```

26 \unexpanded\def\ruledvcenter%
    {\normalvbox\bgroup
     \dontinterfere
     \dowithnextbox
     {\scratchdimen=.5\ht\nextbox
      \advance\scratchdimen by .5\dp\nextbox
      \ht\nextbox=\scratchdimen
      \dp\nextbox=\scratchdimen
      \ruledhbox{\box\nextbox}%
      \egroup}%
     \normalvbox}

```

`\ruledbox`
`\setruledbox`

Of the next two macros the first can be used to precede a box of ones own choice. One can for instance prefix boxes with `\ruledbox` and afterwards — when the macro satisfy the needs — let it to `\relax`.

```
\ruledbox\hbox{What rules do you mean?}
```

The macro `\setruledbox` can be used to directly rule a box.

```
\setruledbox12=\hbox{Who's talking about rules here?}
```

At the cost of some extra macros we can implement a variant that does not need the =, but we stick to:

```

27 \unexpanded\def\ruledbox%
    {\dowithnextbox{\makeruledbox\nextbox\box\nextbox}}

28 \def\setruledbox#1=%
    {\dowithnextbox{\makeruledbox\nextbox\setbox#1=\nextbox}}

```

`\investiga..`
`\investiga..`
`\investiga..`

Before we meet the visualizing macro's, we first implement ourselves some handy utility ones. Just for the sake of efficiency and readability, we introduce some status variables, that tell us a bit more about the registers we use:

```

\ifflexible
\ifzero
\ifnegative
\ifpositive

```

These status variables are set when we call for one of the investigation macros, e.g.

```
\investigateskip\scratchskip
```

We use some dirty trick to check stretchability of $\langle skips \rangle$. Users of these macros are invited to study their exact behavior first. The positive and negative states both include zero and are in fact non-negative (≥ 0) and non-positive (≤ 0).

```

29 \newif\ifflexible
    \newif\ifzero
    \newif\ifnegative
    \newif\ifpositive

30 \def\investigateskip#1%
    {\relax
     \scratchdimen=#1\relax
     \edef\!!stringa{\the\scratchdimen}%
     \edef\!!stringb{\the#1}%
     \ifx\!!stringa\!!stringb \flexiblefalse \else \flexibletrue \fi
     \ifdim#1=\!!zeropoint\relax
       \zerotrue \else
       \zerofalse \fi
     \ifdim#1<\!!zeropoint\relax
       \positivefalse \else
       \positivetrue \fi
     \ifdim#1>\!!zeropoint\relax
       \negativefalse \else
       \negativetrue \fi}

31 \def\investigatecount#1%
    {\relax
     \flexiblefalse
     \ifnum#1=0
       \zerotrue \else
       \zerofalse \fi
     \ifnum#1<0
       \positivefalse \else
       \positivetrue \fi
     \ifnum#1>0
       \negativefalse \else
       \negativetrue \fi}

32 \def\investigatemuskip#1%
    {\relax
     \edef\!!stringa{\the\scratchmuskip}%
     \edef\!!stringb{0mu}%
     \def\!!stringc##1##2\{\##1}%
     \expandafter\edef\expandafter\!!stringc\expandafter
       {\expandafter\!!stringc\!!stringa\}%
     \edef\!!stringd{-}%
     \flexiblefalse
     \ifx\!!stringa\!!stringb
       \zerotrue
       \negativefalse
       \positivefalse
     \else
       \zerofalse
       \ifx\!!stringc\!!stringd

```

```

    \positivefalse
    \negativetrue
  \else
    \positivetrue
    \negativefalse
  \fi
\fi}

```

Now the necessary utility macros are defined, we can make a start with the visualizing ones. The implementation of these macros is a compromise between readability, efficiency of coding and processing speed. Sometimes we do in steps what could have been done in combination, sometimes we use a few boxes more or less than actually needed, and more than once one can find the same piece of rule drawing code twice.

`\ifcentere..`
`\normalvcue` Depending on the context, one can force visual vertical cues being centered along `\hsize` or being put at the current position. Although centering often looks better, we've chosen the second alternative as default. The main reason for doing so is that often when we don't set the `\hsize` ourselves, \TeX takes the value of the surrounding box. As a result the visual cues can migrate outside the current context.

This behavior is accomplished by a small but effective auxiliary macro, which behavior can be influenced by the boolean `\centeredvcue`. By saying

```
\centeredvcuetrue
```

one turns centering on. As said, we turn it off.

```

33 \newif\ifcenteredvcue \centeredvcuefalse
34 \def\normalvcue#1%
    {\normalhbox \ifcenteredvcue to \hsize \fi {\normalhss#1\normalhss}}

```

We could have used the more robust version

```

\def\normalvcue%
  {\normalhbox \ifcenteredvcue to \hsize \fi
   \bgroup\bgroup\normalhss
   \aftergroup\normalhss\aftergroup\egroup
   \let\next=}

```

or the probably best one:

```

\def\normalvcue%
  {\hbox \ifcenteredvcue to \hsize
   \bgroup\bgroup\normalhss
   \aftergroup\normalhss\aftergroup\egroup
   \else
     \bgroup
   \fi
   \let\next=}

```

Because we don't have to preserve *catcodes* and only use small arguments, we stick to the first alternative.

`\testrulew..` We build our visual cues out of rules. At the cost of a much bigger DVI file, this is to be preferred over using characters (1) because we cannot be sure of their availability and (2) because their dimensions are fixed.

As with ruled boxes, we use a $\langle dimension \rangle$ to specify the width of the ruled elements. This dimension defaults to:

```
\testrulewidth=\boxrulewidth
```

Because we prefer whole numbers for specifying the dimensions, we often use even multiples of `\testrulewidth`.

`\visiblest..` A second variable is introduced because of the stretch components of $\langle skips \rangle$. At the cost of some accuracy we can make this stretch visible.

```
\visiblestretchtrue
```

```
35 \newdimen\testrulewidth \testrulewidth=\boxrulewidth
    \newif\ifvisiblestretch \visiblestretchfalse
```

`\ruledhss` We start with the easiest part, the fills. The scheme we follow is *visual filling – going back – normal filling*. Visualizing is implemented using `\cleaders`. Because the $\langle box \rangle$ that follows this command is constructed only once, the `\copy` is not really a prerequisite. We prefer using a `\normalhbox` here instead of a `\hbox`.

```
\ruledhfil
\ruledhfil..
\ruledhfill
\ruledhfil..
```

```
36 \def\setvisiblehfilbox#1\to#2#3#4%
    {\setbox#1=\normalhbox
     {\visualvrule
      \!width#2\testrulewidth
      \!height#3\testrulewidth
      \!depth#4\testrulewidth}%
     \smashbox#1}

37 \def\doruledhfiller#1#2#3#4%
    {#1#2%
     \bgroup
     \dontinterfere
     \dontcomplain
     \setvisiblehfilbox0\to{4}{#3}{#4}%
     \setvisiblehfilbox2\to422%
     \copy0\copy2
     \bgroup
     \setvisiblehfilbox0\to422%
     \cleaders
     \normalhbox to 12\testrulewidth
     {\normalhss\copy0\normalhss}%
     #1%
     \egroup
     \setbox0=\normalhbox
     {\normalhskip-4\testrulewidth\copy0\copy2}%
     \smashbox0
     \box0
     \egroup}
```

The horizontal fillers differ in their boundary visualization. Watch the small dots. Fillers can be combined within reasonable margins.

```
\hss.....test
```

```
\hfil.....test
```

```
\hfill:.....test
```

```
\hfil\hfil.....test.....\hfil
```

The negative counterparts are visualizes, but seldom become visible, apart from their boundaries.

```
\hfilneg.....test
```

```
\hfillneg.....test
```

Although leaders are used for visualizing, they are visualized themselves correctly as the next example shows.

```
.....
```

All five substitutions use the same auxiliary macro. Watch the positive first – negative next approach.

```
38 \unexpanded\def\ruledhss%
    {\doruledhfiller\normalhss\normalhfilneg{0}{0}}
39 \unexpanded\def\ruledhfil%
    {\doruledhfiller\normalhfil\normalhfilneg{10}{-6}}
40 \unexpanded\def\ruledhfill%
    {\doruledhfiller\normalhfill\normalhfillneg{18}{-14}}
41 \unexpanded\def\ruledhfilneg%
    {\doruledhfiller\normalhfilneg\normalhfil{-6}{10}}
42 \unexpanded\def\ruledhfillneg%
    {\doruledhfiller\normalhfillneg\normalhfill{-14}{18}}
```

The vertical mode commands adopt the same visualization scheme, but are implemented in a slightly different way.

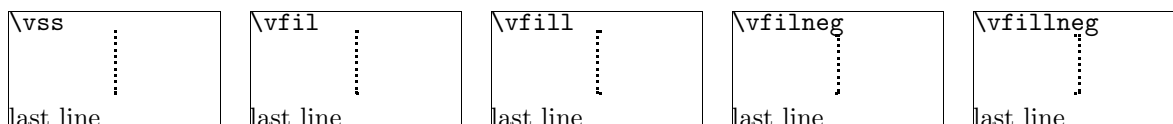
```
\ruledvss
\ruledvfil
\ruledvfil..
\ruledvfill
\ruledvfil..
43 \def\setvisiblevfilbox#1\to#2#3#4%
    {\setbox#1=\normalhbox
     {\visualvrule
      \!!width#2\testrulewidth
      \!!height#3\testrulewidth
      \!!depth#4\testrulewidth}%
     \smashbox#1}%
44 \def\doruledvfiller#1#2#3%
    {#1#2%
     \bgroup
     \dontinterfere
     \dontcomplain
     \normaloffinterlineskip
     \setvisiblevfilbox0\to422%
     \setbox2=\normalvcue
     {\normalhskip -#3\testrulewidth\copy0}%
     \smashbox2}
```

```

\copy2
\bggroup
  \setbox2=\normalvcue
  {\normalhskip -2\testrulewidth\copy0}%
\smashbox2
\copy2
\cleaders
  \normalvbox to 12\testrulewidth
  {\normalvss\copy2\normalvss}%
#1%
\setbox2=\normalvbox
  {\normalvskip-2\testrulewidth\copy2}%
\smashbox2
\box2
\egroup
\box2
\egroup}

```

Because they act the same as their horizontal counterparts we only show a few examples.



Keep in mind that `\vfillneg` is not part of PLAIN $\text{T}_\text{E}\text{X}$, but are mimicked by a macro.

```

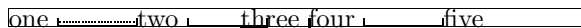
45 \unexpanded\def\ruledvss%
    {\doruledvfiller\normalvss\normalvfilneg{2}}
46 \unexpanded\def\ruledvfil%
    {\doruledvfiller\normalvfil\normalvfilneg{-4}}
47 \unexpanded\def\ruledvfill%
    {\doruledvfiller\normalvfill\normalvfillneg{-12}}
48 \unexpanded\def\ruledvfilneg%
    {\doruledvfiller\normalvfilneg\normalvfil{8}}
49 \unexpanded\def\ruledvfillneg%
    {\doruledvfiller\normalvfillneg\normalvfill{16}}

```

`\ruledhskip` Skips differ from kerns in two important aspects:

- line and pagebreaks are allowed at a skip
- skips can have a positive and/or negative stretchcomponent

Stated a bit different: kerns are fixed skips at which no line or pagebreak can occur. Because skips have a more open character, they are visualized in a open way.



```

one
\hskip +30pt plus 5pt
two
\hskip +30pt
\hskip -10pt plus 5pt
three
\hskip 0pt
four
\hskip +30pt
five

```

When skips have a stretch component, this is visualized by means of a dashed line. Positive skips are on top of the baseline, negative ones are below it. This way we can show the combined results. An alternative visualization of stretch could be drawing the mid line over a length of the stretch, in positive or negative direction.

```

50 \def\doruledhskip%
    {\relax
     \dontinterfere
     \dontcomplain
     \investigateskip\scratchskip
     \ifzero
       \setbox0=\normalhbox
       {\normalhskip-\testrulewidth
        \visualvrule
         \!!width4\testrulewidth
         \!!height16\testrulewidth
         \!!depth16\testrulewidth}%
     \else
       \setbox0=\normalhbox to \ifnegative-\fi\scratchskip
       {\visualvrule
        \!!width2\testrulewidth
        \ifnegative\!!depth\else\!!height\fi16\testrulewidth
        \cleaders
         \visualhrule
         \ifnegative
           \!!depth2\testrulewidth
           \!!height\!!zeropoint
         \else
           \!!height2\testrulewidth
           \!!depth\!!zeropoint
         \fi
        \normalhfill
        \ifflexible
        \normalhskip\ifnegative\else-\fi\scratchskip
        \normalhskip2\testrulewidth
        \cleaders
         \normalhbox
         {\normalhskip 2\testrulewidth

```

```

        \visualvrule
        \!!width2\testrulewidth
        \!!height\ifnegative-7\else9\fi\testrulewidth
        \!!depth\ifnegative9\else-7\fi\testrulewidth
        \normalhskip 2\testrulewidth}%
    \normalhfill
\fi
\visualvrule
\!!width2\testrulewidth
\ifnegative\!!depth\else\!!height\fi16\testrulewidth}%
\setbox0=\normalhbox
{\ifnegative\else\normalhskip-\scratchskip\fi
\box0}%
\fi
\smashbox0%
\ifvisiblestretch \else
\flexiblefalse
\fi
\ifflexible
% breaks ok but small displacements can occur
\skip2=\scratchskip
\advance\skip2 by -1\scratchskip
\divide\skip2 by 2
\advance\scratchskip by -\skip2
\normalhskip\scratchskip
\normalpenalty\!!tenthousand
\box0
\normalhskip\skip2
\else
\normalhskip\scratchskip
\box0
\fi
\egroup}
51 \unexpanded\def\ruledhskip%
{\bgroup
\afterassignment\doruledhskip
\scratchskip=}

```

The visual skip is located at a feasible point. Normally this does not interfere with the normal type-setting process. The next examples show (1) the default behavior, (2) the (not entirely correct) distributed stretch and (3) the way the text is typeset without cues.

```

test test test test test test test test test test test test test test test
test test test test test test test test test test test test test test test
test test test test test test test test test test test test test test test

```

```

test test test test test test test test test test test test test test test
test test test test test test test test test test test test test test test
test test test test test test test test test test test test test test test

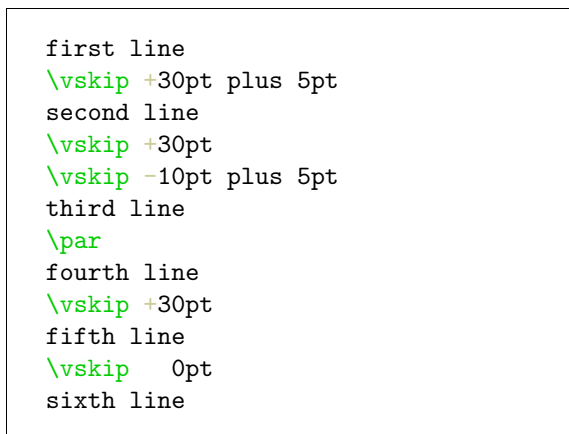
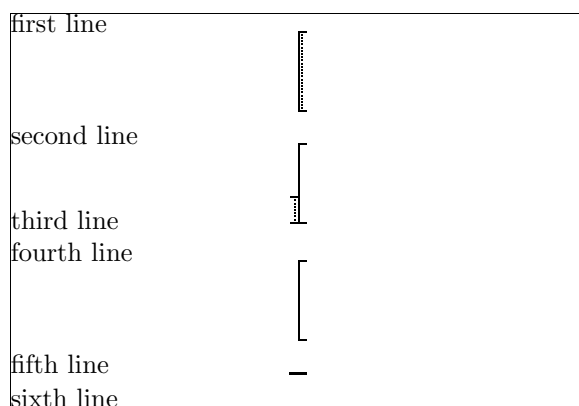
```

```

test test test test test test test test test test test test test test test
test test test test test test test test test test test test test test test
test test test test test test test test test test test test test test test

```


`\ruledvskip` We are less fortunate when implementing the vertical skips. This is a direct result of interference between the boxes that visualize the skip and skip removal at a pagebreak. Normally skips disappear at the top of a page, but not of course when visualized in a `\vbox`. A quite perfect simulation could have been built if we would have had available two more primitives: `\hnop` and `\vnop`. These new primitives could stand for boxes that are visible but are not taken into account in any way. They are there for us, but not for \TeX .



We have to postpone `\prevdepth`. Although this precaution probably is not completely waterproof, it works quite well.

```

52 \def\dodoruledvskip%
    {\nextdepth=\prevdepth
     \dontinterfere
     \dontcomplain
     \normaloffinterlineskip
     \investigateskip\scratchskip
     \ifzero
       \setbox0=\normalvcue
       {\visualvrule
        \!!width32\testrulewidth
        \!!height2\testrulewidth
        \!!depth2\testrulewidth}%
     \else
       \setbox0=\normalvbox to \ifnegative-\fi\scratchskip
       {\visualhrule
        \!!width16\testrulewidth
        \!!height2\testrulewidth
        \ifflexible
        \cleaders
        \normalhbox to 16\testrulewidth
        {\normalhss
         \normalvbox
         {\normalvskip 2\testrulewidth
          \visualhrule
          \!!width2\testrulewidth
          \!!height2\testrulewidth
          \normalvskip 2\testrulewidth}%
         \normalhss}%
        \normalvfill
  
```

```

\else
\normalvfill
\fi
\visualhrule
\!!width16\testrulewidth
\!!height2\testrulewidth}%
\setbox2=\normalvbox to \ht0
{\visualhrule
\!!width2\testrulewidth
\!!height\ht0}%
\ifnegative
\ht0=\!!zeropoint
\setbox0=\normalhbox
{\normalhskip2\testrulewidth % will be improved
\normalhskip-\wd0\box0}%
\fi
\smashbox0%
\smashbox2%
\setbox0=\normalvcue
{\box2\box0}%
\setbox0=\normalvbox
{\ifnegative\normalvskip\scratchskip\fi\box0}%
\smashbox0%
\fi
\ifvisiblestretch
\ifflexible
\skip2=\scratchskip
\advance\skip2 by -1\scratchskip
\divide\skip2 by 2
\advance\scratchskip by -\skip2
\normalvskip\skip2
\fi
\fi
\normalpenalty\!!tenthousand
\box0
\prevdepth=\nextdepth % not \dp0=\nextdepth
\normalvskip\scratchskip}

```

We try to avoid interfering at the top of a page. Of course we only do so when we are in the main vertical list.

```

53 \def\doruledvskip%
{\endgraf % \par
\ifdim\pagegoal=\maxdimen
\ifinner
\dodoruledvskip
\fi
\else
\dodoruledvskip
\fi
\egroup}

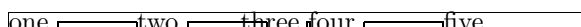
```

```

54 \unexpanded\def\ruledvskip%
    {\bgroup
     \afterassignment\doruledvskip
     \scratchskip=}

```

`\ruledkern` The macros that implement the kerns are a bit more complicated than needed, because they also serve the visualization of glue, our PLAIN defined kerns with stretch or shrink. We've implemented both horizontal and vertical kerns as ruled boxes.



one
<code>\kern +30pt</code>
two
<code>\kern +30pt</code>
<code>\kern -10pt</code>
three
<code>\kern 0pt</code>
four
<code>\kern +30pt</code>
five

Positive and negative kerns are placed on top or below the baseline, so we are able to track their added result. We didn't mention spacings of 0 pt yet. Zero values are visualized a bit different, because we want to see them anyhow.

```

55 \def\doruledhkern%
    {\dontinterfere
     \dontcomplain
     \baselinerulefalse
     \investigateskip\scratchskip
     \boxrulewidth=2\testrulewidth
     \ifzero
       \setbox0=\ruledhbox to 8\testrulewidth
       {\visualvrule
        \!width\!zeropoint
        \!height16\testrulewidth
        \!depth16\testrulewidth}%
       \setbox0=\normalhbox
       {\normalhskip-4\testrulewidth\box0}%
     \else
       \setbox0=\ruledhbox to \ifnegative-\fi\scratchskip
       {\visualvrule
        \!width\!zeropoint
        \ifnegative\!depth\else\!height\fi16\testrulewidth
        \ifflexible
        \normalhskip2\testrulewidth
        \cleaders
        \normalhbox
        {\normalhskip 2\testrulewidth
         \visualvrule
         \!width2\testrulewidth
         \!height\ifnegative-7\else9\fi\testrulewidth

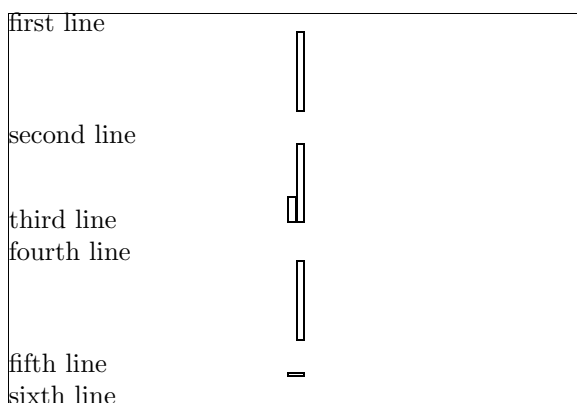
```

```

\!!depth\ifnegative9\else-7\fi\testrulewidth
\normalhskip 2\testrulewidth}%
\normalhfill
\else
\normalhfill
\fi}%
\testrulewidth=2\testrulewidth
\setbox0=\ruledhbox{\box0}% \make...
\fi
\smashbox0%
\normalpenalty\!!tenthousand
\normalhbox to \!!zeropoint
{\ifnegative\normalhskip1\scratchskip\fi
\box0}%
\afterwards\scratchskip
\egroup}
56 \unexpanded\def\ruledhkern#1%
{\bgroup
\let\afterwards=#1\relax
\afterassignment\doruledhkern
\scratchskip=}

```

After having seen the horizontal ones, the vertical kerns will not surprise us. In this example we use `\par` to switch to vertical mode.



```

first line
\par \kern +30pt
second line
\par \kern +30pt
\par \kern -10pt
third line
\par
fourth line
\par \kern +30pt
fifth line
\par \kern 0pt
sixth line

```

Like before, we have to postpone `\prevdepth`. If we leave out this trick, we got ourselves some wrong spacing.

```

57 \def\dodoruledvkern%
{\nextdepth=\prevdepth
\dontinterfere
\dontcomplain
\baselinerulefalse
\normaloffinterlineskip
\investigateskip\scratchskip
\boxrulewidth=2\testrulewidth
\ifzero
\setbox0=\ruledhbox to 32\testrulewidth
{\visualvrule

```

```

    \!!width\!!zeropoint
    \!!height4\testrulewidth
    \!!depth4\testrulewidth}%
\else
  \setbox0=\ruledvbox to \ifnegative-\fi\scratchskip
  {\hsize16\testrulewidth
  \ifflexible
  \cleaders
  \normalhbox to 16\testrulewidth
  {\normalhss
  \normalvbox
  {\normalvskip 2\testrulewidth
  \visualhrule
  \!!width2\testrulewidth
  \!!height2\testrulewidth
  \normalvskip 2\testrulewidth}%
  \normalhss}%
  \normalvfill
  \else
  \visualvrule
  \!!width\!!zeropoint
  \!!height\ifnegative-\fi\scratchskip
  \normalhfill
  \fi}
\fi
\testrulewidth=2\testrulewidth
\setbox0=\ruledvbox{\box0}% \make...
\smashbox0%
\setbox0=\normalvbox
  {\ifnegative\normalvskip\scratchskip\fi
  \normalvcue
  {\ifnegative\normalhskip-16\testrulewidth\fi\box0}}%
\smashbox0%
\normalpenalty\!!tenthousand
\box0
\prevdepth=\nextdepth} % not \dp0=\nextdepth

58 \def\doruledvkern%
  {\ifdim\pagegoal=\maxdimen
  \ifinner
  \dodoruledvkern
  \fi
  \else
  \dodoruledvkern
  \fi
  \afterwards\scratchskip
  \egroup}

59 \unexpanded\def\ruledvkern#1%
  {\bgroup
  \let\afterwards=#1\relax
  \afterassignment\doruledvkern
  \scratchskip=}

```

```

60 \unexpanded\def\ruledkern%
    {\ifvmode
     \expandafter\ruledvkern
    \else
     \expandafter\ruledhkern
    \fi
    \normalkern}

```

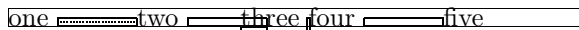
A a bit more TeXnice solution is:

```

\unexpanded\def\ruledkern%
  {\csname ruled\ifvmode v\else h\fi kern\endcsname\normalkern}

```

`\ruledhglue` `\ruledvglue` The non-primitive glue commands are treated as kerns with stretch. This stretch is presented as a dashed line. I have to admit that until now, I've never used these glue commands.



```

one
\hglue +30pt plus 5pt
two
\hglue +30pt
\hglue -10pt plus 5pt
three
\hglue 0pt
four
\hglue +30pt
five

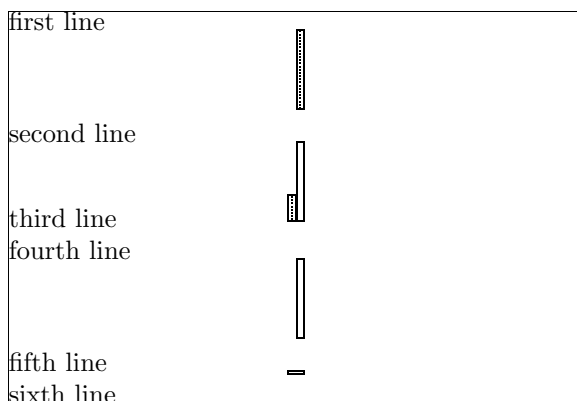
```

```

62 \def\doruledhglue%
    {\leavevmode
     \scratchcounter=\spacefactor
     \visualvrule\!!width\!!zeropoint
     \normalpenalty\!!tenthousand
     \ruledhkern\normalhskip\scratchskip
     \spacefactor=\scratchcounter
     \egroup}

63 \unexpanded\def\ruledhglue%
    {\bgroup
     \afterassignment\doruledhglue\scratchskip=}

```



```

first line
\vglue +30pt plus 5pt
second line
\vglue +30pt
\vglue -10pt plus 5pt
third line
\par
fourth line
\vglue +30pt
fifth line
\vglue 0pt
sixth line

```

```

64 \def\doruledvglue%
    {\endgraf % \par
     \nextdepth=\prevdepth
     \visualhrule\!!height\!!zeropoint
     \normalpenalty\!!tenthousand
     \ruledvkern\normalvskip\scratchskip
     \prevdepth=\nextdepth
     \egroup}

```

```

65 \unexpanded\def\ruledvglue%
    {\bgroup
     \afterassignment\doruledvglue\scratchskip=}

```

`\ruledmkern` Mathematical kerns and skips are specified in mu. This font related unit is incompatible with those of *dimensions* and *skips*. Because in math mode spacing is often a very subtle matter, we've used `\ruledmskip` a very simple, not overloaded way to show them.

```

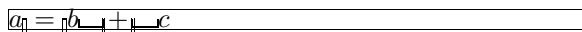
66 \def\dodoruledmkern#1%
    {\dontinterfere
     \dontcomplain
     \setbox0=\normalhbox
     {${\normalmkern\ifnegative-\fi\scratchmuskip$}%
     \setbox0=\normalhbox to \wd0
     {\visualvrule
      \!!height16\testrulewidth
      \!!depth16\testrulewidth
      \!!width\testrulewidth
     \leaders
      \visualhrule
      \!!height\ifpositive16\else-14\fi\testrulewidth
      \!!depth\ifpositive-14\else16\fi\testrulewidth
     \normalhfill
     \ifflexible
     \normalhskip-\wd0
     \leaders
      \visualhrule
      \!!height\testrulewidth
      \!!depth\testrulewidth
    }

```

```

        \normalhfill
    \fi
    \visualvrule
        \!!height16\testrulewidth
        \!!depth16\testrulewidth
        \!!width\testrulewidth}%
\smashbox0%
\ifnegative
    #1\scratchmuskip
    \box0
\else
    \box0
    #1\scratchmuskip
\fi
\egroup}

```



```

$a \mkern3mu = \mkern3mu
b \quad
\mkern-2mu + \mkern-2mu
\quad c$

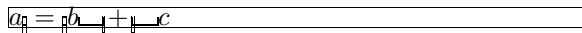
```

```

67 \def\doruledmkern%
    {\investigatemuskip\scratchmuskip
    \flexiblefalse
    \dodoruledmkern\normalmkern}

68 \unexpanded\def\ruledmkern%
    {\bgroup
    \afterassignment\doruledmkern\scratchmuskip=}

```



```

$a \mskip3mu = \mskip3mu
b \quad
\mskip-2mu + \mskip-2mu
\quad c$

```

```

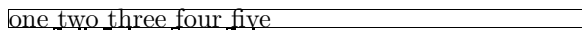
69 \def\doruledmskip%
    {\investigatemuskip\scratchmuskip
    \flexibletrue
    \dodoruledmkern\normalmskip}

70 \unexpanded\def\ruledmskip%
    {\bgroup
    \afterassignment\doruledmskip\scratchmuskip=}

```

`\penalty` After presenting fills, skip, kerns and glue we've come to see penalties. In the first implementation — most of the time needed to develop this set of macros went into testing different types of visualization — penalties were mere small blocks with one black half, depending on the sign. This most recent version also gives an indication of the amount of penalty. Penalties can go from less than -10000 to

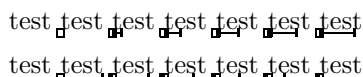
over +10000, and their behavior is somewhat non-linear, with some values having special meanings. We therefore decided not to use its value for a linear indicator.



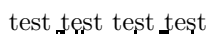
```

one
\penalty +100
two
\penalty +100
\penalty -100
three
\penalty 0
four
\penalty +100
five
    
```

The small sticks at the side of the penalty indicate it size. The next example shows the positive and negative penalties of 0, 1, 10, 100, 1000 and 10000.



This way stacked penalties of different severance can be shown in combination.



```

71 \def\setruledpenaltybox#1#2#3#4#5#6%
    {\setbox#1=\normalhbox
     {\ifnum#2=0 \else
      \ifnum#2>0
        \def\sign{+}%
      \else
        \def\sign{-}%
      \fi
      \dimen0=\ifnum\sign#2>9999
        28\else
        \ifnum\sign#2>999
          22\else
          \ifnum\sign#2>99
            16\else
            \ifnum\sign#2>9
              10\else
              4
            \fi\fi\fi\fi \testrulewidth
      \ifnum#2<0
        \normalhskip-\dimen0
        \normalhskip-2\testrulewidth
        \visualvrule
        \!!width2\testrulewidth
        \!!height#3\testrulewidth
        \!!depth#4\testrulewidth
      \fi
      \visualvrule
      \!!width\dimen0
      \!!height#5\testrulewidth
    }
    
```

```





        \!!depth#6\testrulewidth
    \ifnum#2>0
        \visualvrule
            \!!width2\testrulewidth
            \!!height#3\testrulewidth
            \!!depth#4\testrulewidth
        \fi
    \fi}%
\smashbox#1}

72 \def\doruledhpenalty%
    {\dontinterfere
    \dontcomplain
    \investigatecount\scratchcounter
    \testrulewidth=2\testrulewidth
    \boxrulewidth=\testrulewidth
    \setbox0=\ruledhbox to 8\testrulewidth
    {\ifnegative\else\normalhss\fi
    \visualvrule
        \!!depth8\testrulewidth
        \!!width\ifzero0\else4\fi\testrulewidth
    \ifpositive\else\normalhss\fi}%
    \setruledpenaltybox{2}{\scratchcounter}{0}{8}{-3.5}{4.5}%
    \normalpenalty\!!tenthousand
    \setbox0=\normalhbox
    {\normalhskip-4\testrulewidth
    \ifnegative
        \box2\box0
    \else
        \box0\box2
    \fi}%
    \smashbox0%
    \box0
    \normalpenalty\scratchcounter
    \egroup}

73 \unexpanded\def\ruledhpenalty%
    {\bgroup
    \afterassignment\doruledhpenalty
    \scratchcounter=}

```

The size of a vertical penalty is also shown on the horizontal axis. This way there is less interference with the often preceding or following skips and kerns.

first line	
second line	
third line	
fourth line	
fifth line	

```

first line
\par \penalty +100
second line
\par \penalty +100
\par \penalty -100
third line
\par \penalty 0
fourth line
\par \penalty +100
fifth line

```

```

74 \def\doruledvpenalty%
    {\ifdim\pagegoal=\maxdimen
    \else
    \nextdepth=\prevdepth
    \dontinterfere
    \dontcomplain
    \investigatecount\scratchcounter
    \testrulewidth=2\testrulewidth
    \boxrulewidth=\testrulewidth
    \setbox0=\ruledhbox
    {\visualvrule
    \!!height4\testrulewidth
    \!!depth4\testrulewidth
    \!!width\!!zeropoint
    \visualvrule
    \!!height\ifnegative.5\else4\fi\testrulewidth
    \!!depth\ifpositive.5\else4\fi\testrulewidth
    \!!width8\testrulewidth}%
    \setruledpenaltybox{2}{\scratchcounter}{4}{4}{.5}{.5}%
    \setbox0=\normalhbox
    {\normalhskip-4\testrulewidth
    \ifnegative
    \box2\box0
    \else
    \box0\box2
    \fi
    \normalhss}%
    \smashbox0%
    \normalpenalty\!!tenthousand
    \nointerlineskip
    \dp0=\nextdepth % not \prevdepth=\nextdepth
    \normalvbox
    {\normalvcue{\box0}}%
    \fi
    \normalpenalty\scratchcounter
    \egroup}

```

```

75 \unexpanded\def\ruledvpenalty%
    {\bgroup
     \afterassignment\doruledvpenalty
     \scratchcounter=}
76 \unexpanded\def\ruledpenalty%
    {\ifvmode
     \expandafter\ruledvpenalty
     \else
     \expandafter\ruledhpenalty
     \fi}

```

At the cost of some more tokens, a bit more clever implementation would be:

```

\unexpanded\def\ruledpenalty%
  {\csname ruled\ifvmode v\else h\fi penalty\endcsname}

```

For those who want to manipulate the visual cues in detail, we have grouped them.

```

\showfiles
\dontshowf..
\showboxes
\dontshowb..
\showskips
\dontshows..
\showpenal..
\dontshowp..
78
\def\showfiles%
  {\let\hss      = \ruledhss
   \let\hfil     = \ruledhfil
   \let\hfill    = \ruledhfill
   \let\hfilneg  = \ruledhfilneg
   \let\hfillneg = \ruledhfillneg
   \let\vss      = \ruledvss
   \let\vfil     = \ruledvfil
   \let\vfill    = \ruledvfill
   \let\vfilneg  = \ruledvfilneg
   \let\vfillneg = \ruledvfillneg}
79
\def\dontshowfiles%
  {\let\hss      = \normalhss
   \let\hfil     = \normalhfil
   \let\hfill    = \normalhfill
   \let\hfilneg  = \normalhfilneg
   \let\hfillneg = \normalhfillneg
   \let\vss      = \normalvss
   \let\vfil     = \normalvfil
   \let\vfill    = \normalvfill
   \let\vfilneg  = \normalvfilneg
   \let\vfillneg = \normalvfillneg}
80
\def\showboxes%
  {\baselineruletrue
   \let\hbox     = \ruledhbox
   \let\vbox     = \ruledvbox
   \let\vtop    = \ruledvtop
   \let\vcenter  = \ruledvcenter}
81
\def\dontshowboxes%
  {\let\hbox     = \normalhbox
   \let\vbox     = \normalvbox
   \let\vtop    = \normalvtop
   \let\vcenter  = \normalvcenter}

```

```

82 \def\showskips%
    {\let\hskip = \ruledhskip
     \let\vskip = \ruledvskip
     \let\kern = \ruledkern
     \let\mskip = \ruledmskip
     \let\mkern = \ruledmkern
     \let\hglue = \ruledhglue
     \let\vglue = \ruledvglue}

83 \def\dontshowskips%
    {\let\hskip = \normalhskip
     \let\vskip = \normalvskip
     \let\kern = \normalkern
     \let\mskip = \normalmskip
     \let\mkern = \normalmkern
     \let\hglue = \normalhglue
     \let\vglue = \normalvglue}

84 \def\showpenalties%
    {\let\penalty = \ruledpenalty}

85 \def\dontshowpenalties%
    {\let\penalty = \normalpenalty}

```

`\showingco..` All these nice options come together in two macros. The first one turns the options on, the second
`\showcompo..` turns them off. Both macros only do their job when we are actually showing the composition.
`\dontshowc..`

```

    \showingcompositiontrue
    \showcomposition

```

Because the output routine can do tricky things, like multiple column typesetting and manipulation of the pagebody, shifting things around and so on, the macro `\dontshowcomposition` best can be called when we enter this routine. Too much visual cues just don't make sense. In `ConTeXt` this has been taken care of.

```

86 \newif\ifshowingcomposition

87 \def\showcomposition%
    {\ifshowingcomposition
     \showfiles
     \showboxes
     \showskips
     \showpenalties
     \fi}

88 \def\dontshowcomposition%
    {\ifshowingcomposition
     \dontshowfiles
     \dontshowboxes
     \dontshowskips
     \dontshowpenalties
     \fi}

```

`\showmakeup` Just to make things even more easy, we have defined:

```

\defaulttete..
    \showmakeup

```

For the sake of those who don't (yet) use ConTEXt we preset `\defaultttestrulewidth` to the already set value. Otherwise we default to a bodyfontsize related value.

```
\def\defaultttestrulewidth{.2pt}
```

Beware, it's a macro not a *dimension*.

```
89 \ifx\korpsgrootte\undefined
    \edef\defaultttestrulewidth{\the\testrulewidth}
    \else
    \def\defaultttestrulewidth{.02\korpsgrootte} % still dutch
    \fi

90 \def\showmakeup%
    {\testrulewidth=\defaultttestrulewidth
    \showingcompositiontrue
    \showcomposition}

91 \protect
```

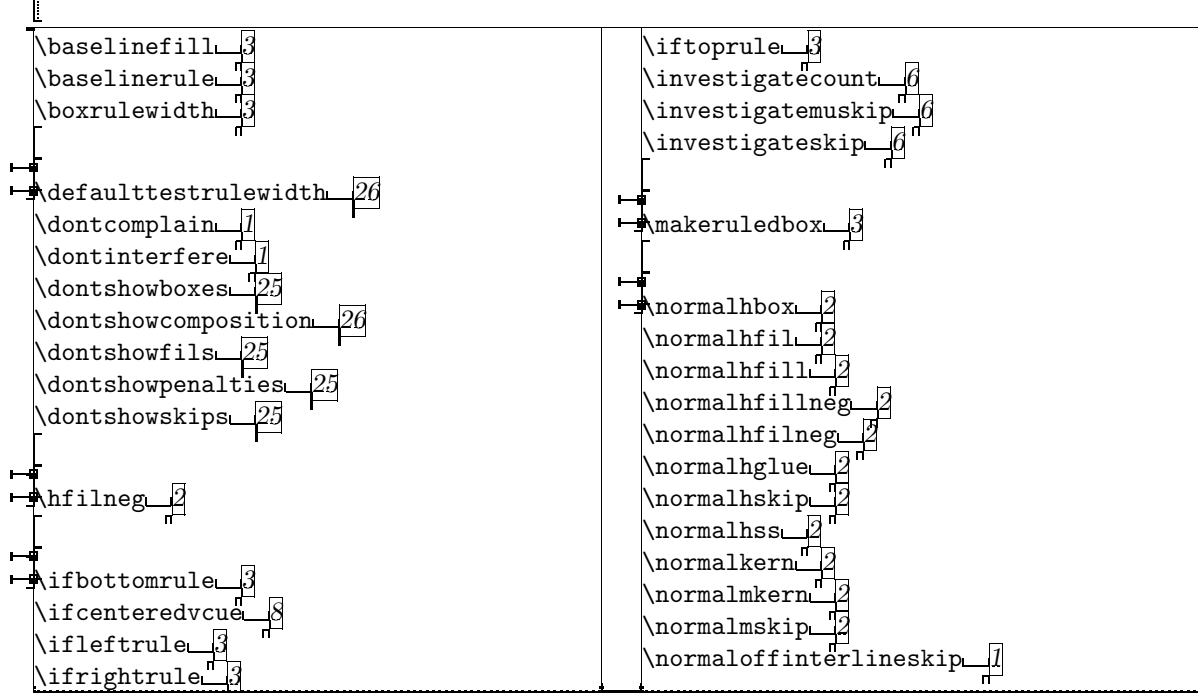
Lets end with some more advanced examples. Definitions and enumerations come in many flavors. The next one for instance is defined as:

```
\definedescription[test] [place=left,hang=3,width=6em]
```

When applied to some text, this would look like:

visual..... I would be very pleased if TEX had two more primitives: `\vno`p and `\hno`p. Both should
debugger..... act and show up as normal boxes, but stay invisible for TEX when it's doing calculations.
 The `\vno`p for instance should not interact with the internal mechanism responsible for the disappearing skips, kerns and penalties at a pagebreak. As long as we don't have these two boxtypes, visual debugging will never be perfect.

The index to this section looks like:



<code>\normalpenalty</code>	2	<code>\ruledvbox</code>	5
<code>\normalvbox</code>	2	<code>\ruledvcenter</code>	5
<code>\normalvcue</code>	8	<code>\ruledvfil</code>	10
<code>\normalvfil</code>	2	<code>\ruledvfill</code>	10
<code>\normalvfill</code>	2	<code>\ruledvfillneg</code>	10
<code>\normalvfillneg</code>	2	<code>\ruledvfilneg</code>	10
<code>\normalvfilneg</code>	2	<code>\ruledvglue</code>	19
<code>\normalvglue</code>	2	<code>\ruledvskip</code>	14
<code>\normalvskip</code>	2	<code>\ruledvss</code>	10
<code>\normalvss</code>	2	<code>\ruledvtop</code>	5
<code>\normalvtop</code>	2		
<code>\penalty</code>	21	<code>\setruledbox</code>	6
<code>\ruledbox</code>	6	<code>\showboxes</code>	25
<code>\ruledhbox</code>	5	<code>\showcomposition</code>	26
<code>\ruledhfil</code>	9	<code>\showfils</code>	25
<code>\ruledhfill</code>	9	<code>\showingcomposition</code>	26
<code>\ruledhfillneg</code>	9	<code>\showmakeup</code>	26
<code>\ruledhfilneg</code>	9	<code>\showpenalties</code>	25
<code>\ruledhglue</code>	19	<code>\showskips</code>	25
<code>\ruledhskip</code>	11	<code>\testrulewidth</code>	8
<code>\ruledhss</code>	9	<code>\vfilneg</code>	2
<code>\ruledkern</code>	16	<code>\visiblestretch</code>	9
<code>\ruledmkern</code>	20		
<code>\ruledmskip</code>	20		

Although not impressive examples or typesetting, both show us how and where things happen. When somehow the last lines in this two column index don't align, then this is due to some still unknown interference.

92 `\endinput`

<code>\baselinefill</code>	3	<code>\normalvfilneg</code>	2
<code>\baselinerule</code>	3	<code>\normalvglue</code>	2
<code>\boxrulewidth</code>	3	<code>\normalvskip</code>	2
		<code>\normalvss</code>	2
		<code>\normalvtop</code>	2
<code>\defaultttestrulewidth</code>	26		
<code>\dontcomplain</code>	1	<code>\penalty</code>	21
<code>\dontinterfere</code>	1		
<code>\dontshowboxes</code>	25	<code>\ruledbox</code>	6
<code>\dontshowcomposition</code>	26	<code>\ruledhbox</code>	5
<code>\dontshowfils</code>	25	<code>\ruledhfil</code>	9
<code>\dontshowpenalties</code>	25	<code>\ruledhfill</code>	9
<code>\dontshowskips</code>	25	<code>\ruledhfillneg</code>	9
		<code>\ruledhfilneg</code>	9
<code>\hfilneg</code>	2	<code>\ruledhglue</code>	19
		<code>\ruledhskip</code>	11
<code>\ifbottomrule</code>	3	<code>\ruledhss</code>	9
<code>\ifcenteredvcue</code>	8	<code>\ruledkern</code>	16
<code>\iflefttrule</code>	3	<code>\ruledmkern</code>	20
<code>\ifrighttrule</code>	3	<code>\ruledmskip</code>	20
<code>\iftoprule</code>	3	<code>\ruledvbox</code>	5
<code>\investigatecount</code>	6	<code>\ruledvcenter</code>	5
<code>\investigatemuskip</code>	6	<code>\ruledvfil</code>	10
<code>\investigateskip</code>	6	<code>\ruledvfill</code>	10
		<code>\ruledvfillneg</code>	10
<code>\makeruledbox</code>	3	<code>\ruledvfilneg</code>	10
		<code>\ruledvglue</code>	19
<code>\normalhbox</code>	2	<code>\ruledvskip</code>	14
<code>\normalhfil</code>	2	<code>\ruledvss</code>	10
<code>\normalhfill</code>	2	<code>\ruledvtop</code>	5
<code>\normalhfillneg</code>	2		
<code>\normalhfilneg</code>	2	<code>\setruledbox</code>	6
<code>\normalhglue</code>	2	<code>\showboxes</code>	25
<code>\normalhskip</code>	2	<code>\showcomposition</code>	26
<code>\normalhss</code>	2	<code>\showfils</code>	25
<code>\normalkern</code>	2	<code>\showingcomposition</code>	26
<code>\normalmkern</code>	2	<code>\showmakeup</code>	26
<code>\normalmskip</code>	2	<code>\showpenalties</code>	25
<code>\normaloffinterlineskip</code>	1	<code>\showskips</code>	25
<code>\normalpenalty</code>	2		
<code>\normalvbox</code>	2	<code>\testrulewidth</code>	8
<code>\normalvcue</code>	8		
<code>\normalvfil</code>	2	<code>\vfilneg</code>	2
<code>\normalvfill</code>	2	<code>\visiblestretch</code>	9
<code>\normalvfillneg</code>	2		

$\mathcal{N}\mathcal{T}\mathcal{S}$: a New Typesetting System

Karel Skoupy

Faculty of Informatics

Botanická 68a

602 00 Brno, Czech Republic

Phone: +420-5-752-040

skoupy@informatics.muni.cz

Abstract

$\mathcal{N}\mathcal{T}\mathcal{S}$ represents one possible radical approach to the idea of making a successor for \TeX . Its underlying theme is the complete re-implementation of \TeX : The Program in Java. The first version will be compatible with \TeX but the structure of the new program will be as open and modular as possible. At the time when \TeX : The Program was written, computer performance and programming technology were very limited in comparison with today. Object orientation and the many other modern features of Java will make many problems easier to solve and will allow for far greater generality. Polymorphic objects will handle different font or output formats directly without affecting the rest of the system. The new implementation will provide a platform on which further experimentation can be conducted; such experimentation may aim, for example, to improve typographic quality (e.g. page break optimization) and/or facilitate integration with other systems.

When considering the future of \TeX and its potential successor(s), there were five options available [9]. They ranged from the most conservative—to leave \TeX exactly as it is—to the most radical—to design quite a new typesetting system for the next century. The $\mathcal{N}\mathcal{T}\mathcal{S}$ project started with a relatively conservative approach to add some extensions and enhancements to the current \TeX which resulted in $\varepsilon\text{-}\text{\TeX}$. It was agreed that the design of a new system from scratch is worthy but it had to be postponed until the $\mathcal{N}\mathcal{T}\mathcal{S}$ project had adequate financial resources.

The funds allocated to the $\mathcal{N}\mathcal{T}\mathcal{S}$ project by DANTE e.V. enabled the radical approach to $\mathcal{N}\mathcal{T}\mathcal{S}$ to get off the ground. The first meeting of the $\mathcal{N}\mathcal{T}\mathcal{S}$ group was held in Zeuthen (during a regular DANTE meeting) between October 8–11 1997. Although the current author (who was to be employed as a full-time programmer) was busy with another project, it was planned that he could start work on $\mathcal{N}\mathcal{T}\mathcal{S}$ near the beginning of 1998. It of course did not prevent the working group from thinking about the problem.

Fundamental Desiderata

It had been already decided that $\mathcal{N}\mathcal{T}\mathcal{S}$ should not be designed from scratch entirely— \TeX : The Program (or more precisely $\varepsilon\text{-}\text{\TeX}$) was chosen as the starting point. What had to be done from scratch was a com-

plete re-implementation. Such a re-implementation has to be compatible with current \TeX , and ideally it should pass the TRIP test (unless there are really good reasons for not passing it). This constraint has its advantages. As \TeX is considered (by the working group) to be the best typesetting system on the world, compatibility will prove that $\mathcal{N}\mathcal{T}\mathcal{S}$ is at least as good as its predecessor. Also \TeX : The Program is an extremely well designed application full of inspiring ideas and it would not be wise to drop them.

On the other hand the structure of the new system should be made as open as possible. It should be partitioned into relatively independent modules with well defined interface. It will eventually allow for great changes by only local intervention to a particular module without affecting the rest of the system. Although the functionality will be compatible with \TeX , both the structure of the program and the data structures should be ready for such changes and extensions that are likely to be desirable for $\mathcal{N}\mathcal{T}\mathcal{S}$ (Properties of a potential new typesetting system were discussed in [7, 3, 2]). For this purpose an object oriented design seemed to be the most suitable.

Implementation language The first task of the group was to choose a programming language for the implementation. Several high level languages

for fast prototyping had been taken into account in the past. Eventually three well known object oriented languages were considered: Common Lisp Object System (CLOS), C++ and Java. Although each of these languages has its specific advantages, after careful comparison of them Java was chosen.

Some of its characteristics follow. Java as the youngest of these languages could learn from them. It has a very advanced implementation of objects and their interfaces are orthogonal to the class hierarchy. Classes are compiled into so-called byte code and can be dynamically combined at run time. The built in garbage collector is very convenient and prevents the programmer from causing many memory violation errors. Types and their checking can also catch many errors, from trivial to serious. Exception handling requires precise declarations which prevent forgetting of boundary situations. Java is completely portable even at the level of compiled byte code. Although the standard interpretation by the Java Virtual Machine is not as efficient as machine code, there are compilers to native code available. There is also a wide range of standard libraries for networking, graphic, graphical user interfaces and others.

One very important point is Java's Internet awareness. You can imagine that you will be able to (automatically) download new modules and $\mathcal{N}\mathcal{T}\mathcal{S}$ plug-ins, share fonts and input texts over the Internet and use JavaBeans¹ to tailor the system for your needs interactively. We also anticipate good support for Java in future. This is now very popular and many new applications are being implemented in it. It is quite possible that a Java interpreter will eventually be burned onto a silicon chip at some point in the not-too distant future.

Design of $\mathcal{N}\mathcal{T}\mathcal{S}$

During the Spring of 1998 there were two more meetings of the $\mathcal{N}\mathcal{T}\mathcal{S}$ working group. Mainly discussed were the features of $\mathcal{N}\mathcal{T}\mathcal{S}$ which will not be implemented in the first version but which the design has to anticipate. Gradually the specification for the first version was set.

The first task of real work on the implementation was to make a design with proposed structure of the $\mathcal{N}\mathcal{T}\mathcal{S}$. It was ready by May 2 1998 and reviewed by Philip Taylor and Jiří Zlatuška. The presentation of the main design decisions will follow. We do not want to go into very implementation-specific details, it will instead be a rather informal description

¹ JavaBeans is a component architecture that helps independent vendors write classes that can be treated as components of larger systems assembled by users.

of things which might be of interest. Comments are of course welcome.

General notes The specification for re-implementation is rather simple. The main source of information is TEX : The Program itself. The task is not to design something with a different philosophy, it is rather to take used principles and make them more open and general. It seems that there were two main constraints at the time of the creation of TEX — the low performance of machines and the programming technology available.

The first problem related to these constraints is memory management. On the one hand the memories of computers at that time were very small compared with today, whilst on the other hand there was probably no standard support for dynamic memory management. Knuth decided to create his own memory management based on preallocated buffers. Today we are much less constrained in using memory. The physical memories of today's computers are much bigger and current operating systems provide even larger virtual memories.

The second problem was concerned with the data structures. It is apparent from the source code that Knuth tried to use memory in the most effective way. He did not accept the standard Pascal records and pointers and rather used preallocated arrays in a very compact way. This was good for performance but it would be very painful to add new data structures to existing scheme. The symbolic names for structure items are maintained using WEB macros and therefore their types are not distinguished and checked by compiler.

Fortunately Java provides us with very advanced memory management with garbage collection. It is very natural to accept Java objects as data structures, not to take special care about memory at all and not to impose any explicit limits to the size of internal buffers.

WEB preprocessor We believe that Knuth took the best programming language available for his purpose at the time. But even standard Pascal was not enough and he made the WEB preprocessor mainly for three reasons: literate programming, macro definitions and rearranging the source text.

Pascal serves well for the structured programming paradigm. But if you peek into TEX : The Program you can recognize steps towards an object oriented paradigm supported by WEB . The code dealing with particular data structures — mainly the relevant parts of switches in general procedures as (for example) symbolic printing or of course the `main_control` — is usually gathered in one place. In an

object oriented language such processing is implemented by virtual methods of objects and there is no need for rearranging the order of source code and for most of the switches any more.

The need for macro definitions is very much reduced, too. In C++ the usage of macros was significantly replaced by templates. Unfortunately standard Java does not provide any of these facilities (there are other implementations which have templates and operator overloading as an extension, e.g. jump²). This should not impact too badly on $\mathcal{N}\mathcal{T}\mathcal{S}$: templates are mostly used in general frameworks.

We certainly want the new program to be well documented, but we are still not concerned to make a book of it. Java has its own source documentation facility. The documentation is in the form of pragmatic comments and is supposed to be in HTML or a similar SGML-based format. It seems sufficient to us now. Well, it may change in future but in such a case it is not necessary to change the program code itself.

Character set, fonts and hyphenation. In current $\mathbb{T}\mathbb{E}\mathbb{X}$ the input encoding, hyphenation pattern encoding and font encoding must be the same. Also there are only 256 character codes. It may be sufficient for English but it makes usage of $\mathbb{T}\mathbb{E}\mathbb{X}$ harder for other languages. There are ways around it (virtual fonts, pattern sources independent of character encoding), the thing just can be made much easier.

Fonts and hyphenation tables will be implemented as object providing methods under a well defined interface. It is possible that objects for different formats provide the same interface and for example PostScript font metrics can be used directly. We can make the interface to such objects independent of encoding using character names. Internally the characters will be represented by numbers but there will be a module for mapping to external names. In case the font does not know the names (pk fonts), the process will default to numeric codes. The codes can possibly be re-mapped by tables for different encodings and independent of fonts.

Diagnostics. $\mathbb{T}\mathbb{E}\mathbb{X}$ uses terminal and log files for diagnostic purposes. It can be made much more general by polymorphic log objects. Some users might prefer some windowed output (strange but possible), but mainly it can be used by some wrapping application having $\mathcal{N}\mathcal{T}\mathcal{S}$ as a processor inside.

Basic types. Numbers, dimens and glues will be in $\mathcal{N}\mathcal{T}\mathcal{S}$ as well. Their semantics will be the same

² jump is a free Java compiler with mentioned capabilities, see: <http://ourworld.compuserve.com/homepages/DeHoeffner/jump.htm>

as in $\mathbb{T}\mathbb{E}\mathbb{X}$. Maybe in future we will add other types for some extended typesetting tasks.

Language. In $\mathbb{T}\mathbb{E}\mathbb{X}$ the input characters are transformed to tokens and they are after macro expansion transformed to primitive commands which are handled by the chief executive. In $\mathcal{N}\mathcal{T}\mathcal{S}$ the process will be similar but our aim is to separate different layers as much as possible. The input, tokenization and macro expansion will be a straightforward re-implementation using objects.

Although in $\mathbb{T}\mathbb{E}\mathbb{X}$ there are dependencies between the macro language and typesetting (the dimensions of boxes in registers, named typesetting parameters, output routine) we will try to make these dependencies clear and handled via well defined interface. This might allow us to provide sometime in the future an alternative input language (procedural, object oriented, ...) without any incompatible change (extensions might be necessary) to the typesetting engine driven by primitive commands.

Modes. The meaning of a primitive command may depend on the current mode. There are three main modes (vertical, horizontal, maths) in $\mathbb{T}\mathbb{E}\mathbb{X}$ each with two submodes. In $\mathcal{N}\mathcal{T}\mathcal{S}$ it will be much more general. The modes will not be just codes but polymorphic objects. They will provide methods such as adding the next character, kern or glue and so on. In some cases the method will issue an error message ("You can't ... in ... mode"), in other cases it will push another object onto the stack of modes. Each command will know how to apply itself when meeting with a mode.

This will allow for making new specialized modes derived from existing ones. We will try (for example) to implement alignments in this way. Commands not aware of any specialization will pass in the usual way but specialized commands can test the current mode and invoke some extra method not included in the base mode interface. This approach may make non-textual modes for chemical formulæ, pictures or music notes more easy and efficient.

Lists and boxes. The objects in lists will be polymorphic so the formatting algorithms will handle each object uniformly. They will invoke only the appropriate methods to get information about size, stretchability, ...

In $\mathbb{T}\mathbb{E}\mathbb{X}$ the lists are static once created. After breaking a paragraph into lines there is no easy way to reformat it. Also the parameters used by the paragraph breaking algorithm (such as `\tolerance` or `\hyphenpenalty`) are lost after processing. In

$\mathcal{N}\mathcal{T}\mathcal{S}$ the lists will remain dynamic and keep all the information necessary for reformatting later.

One application might be a WYSIWYG interactive program which needs to reformat a paragraph after a user's change to it. We do not plan such a program to be part of $\mathcal{N}\mathcal{T}\mathcal{S}$ but $\mathcal{N}\mathcal{T}\mathcal{S}$ might be used by someone else as an internal engine and we will try to make it possible. Provided that the modules of $\mathcal{N}\mathcal{T}\mathcal{S}$ are independent, they can be used by such applications directly. In such case the applications can also employ lists as their own data structures.

But there is a more important reason from the point of view of high quality typesetting—global page break optimization. Actually this was one of the main challenges to $\mathcal{N}\mathcal{T}\mathcal{S}$. For solving this problem we will need to keep the whole main vertical list and try to find a satisfactory (or optimal) sequence of page breaks. Then reformatting of paragraphs might be helpful. If we allow the page layout (`\hsize`) to change for different pages it will become a necessity. Even without global optimization the changing page layout is a problem and more general shapes of paragraphs (than defined by `\hsize` and `\parshape`) will be useful.

Maths formulæ. \TeX is excellent in the typesetting of mathematics. The formulæ have their own representation which is transformed to usual boxes when contributed to the parent list. The difference in $\mathcal{N}\mathcal{T}\mathcal{S}$ approach is that the objects for formulæ will be descendants of the same base class as the ordinary text. It will not need transformation and will be kept dynamically in the same way as lists.

Output. You might guess that also the output will be shipped out by a polymorphic object. It will have methods for setting characters, rules, images or some graphic objects. The objects in the list will know which method to invoke. One implementation can produce DVI, others can generate PostScript or PDF.

Algorithm parameterization. \TeX 's algorithms are parameterized in many ways. There are a lot of numeric parameters to the paragraph breaking algorithm, page output and page breaking is influenced by a user defined output routine. We can make parameterization more general. Beside an enriched set of variable parameters we will prepare void virtual methods which can (for example) add extra demerits to two consecutive broken lines in a paragraph or to a whole sequence of potential line breaks. By supplying some smart code to such methods, it will be possible to avoid "rivers" and other subtleties not solved by \TeX . Such parameterization will be done by making a specialized version of $\mathcal{N}\mathcal{T}\mathcal{S}$ with

overridden methods or by more convenient plug-ins. The possibility of giving access to internal lists in the input language and user supplied methods from the input file should be discussed.

Conclusion

We described the current state of the initial phase of the $\mathcal{N}\mathcal{T}\mathcal{S}$ project aiming at re-implementing \TeX in Java so that the internal structure of the program will allow for experiments and modifications of the algorithms used or the actions taken when typesetting using \TeX . Basic design decisions behind the choice of the implementation language and the object-oriented programming paradigm have been exposed and the overall structure of the resulting program has been outlined.

The first version of the $\mathcal{N}\mathcal{T}\mathcal{S}$ is now under development and should be available by the beginning of 1999.

The author wishes to express thanks to Don Knuth for making \TeX available, the $\mathcal{N}\mathcal{T}\mathcal{S}$ group for fruitful discussions, contributors to the NTS-L list for many interesting ideas, and DANTE e.V. for continuing support in this endeavor.

References

- [1] Ken Arnold, James Gosling: "The Java Programming Language, Second Edition", Addison-Wesley Publishing Company, Reading, Mass., December 1997.
- [2] Michael Barr: " \TeX wish list", in *TUGboat*, Vol. 13, No. 2, pp. 223–226, July 1992.
- [3] Nelson H. F. Beebe: "Comments on the future of \TeX and METAFONT", in *TUGboat*, Vol. 11, No. 4, pp. 490–494, November 1990.
- [4] Roger Hunter: "A future for \TeX ", in *TUGboat*, Vol. 14, No. 3, pp. 183–186, October 1993.
- [5] Donald E. Knuth: "The future of \TeX and METAFONT", in *TUGboat*, Vol. 11, No. 4, pp. 489–489, November 1990.
- [6] Donald E. Knuth: " \TeX : The Program", Addison-Wesley Publishing Company, Reading, Mass., 1986.
- [7] Frank Mittelbach: "E- \TeX : Guidelines for future \TeX ", in *TUGboat*, Vol. 11, No. 3, pp. 337–345, September 1990.
- [8] Philip Taylor: " \TeX : The next generation", in *TUGboat*, Vol. 13, No. 2, pp. 138–138, July 1992.
- [9] Philip Taylor: "The future of \TeX ", in *TUGboat*, Vol. 13, No. 4, pp. 433–442, December 1992.

- [10] Philip Taylor: “NTS: the future of \TeX ”, in *TUGboat*, Vol. 14, No. 3, pp. 177–182, October 1993.
- [11] Philip Taylor: “NTS update”, in *TUGboat*, Vol. 14, No. 4, pp. 381–382, December 1993.
- [12] Philip Taylor: “Report of the 2nd meeting of the NTS group, February 1994”, in *TUGboat*, Vol. 15, No. 2, pp. 96–97, June 1994.
- [13] Philip Taylor: “Minutes of the NTS meeting held at Lindau on October 11/12th 1994”, in *TUGboat*, Vol. 15, No. 4, pp. 434–437, December 1994.
- [14] Philip Taylor: “NTS & $\varepsilon\text{-}\TeX$: a status report”, in *TUGboat*, Vol. 18, No. 1, pp. 6–12, March 1997.
- [15] Zlatuška, Jiří (ed): *Euro \TeX '92 Proceedings*, pp. 235–254, September 1992. Published by $\text{\textcircled{C}}\text{TUG}$, Czechoslovak \TeX Users Group, ISBN 80-210-0480-0.

TeX in 2003: Part I

Propositions and Conjectures on the Future of TeX*

NTG TeX future working group

P.O. Box 394
1740 AJ Schagen
The Netherlands
ntg-toekomsttex@ntg.nl
http://www.ntg.nl

Introduction

In the last year, there has been a lively discussion within the Dutch TeX Users Group about the future of TeX. This discussion was initialized by a couple of posts to the TEX-NL e-mail list by Hans Hagen and Taco Hoekwater, but it soon spread to a much larger group of correspondents.

Eventually, this resulted in a meeting of the most interested people in December 1997. The current articles are a re-working of the long-term proposals and requests formulated by this group of people. The short-term requests were passed on to the ε -TeX team.

Our views on current work

At the moment, there are at least three distinct projects available to current TeX users that are working to extend TeX: Ω , PdfTeX and ε -TeX.

The first two of these are in a sense niche products: If you don't need either non-latin language typesetting or PDF output, there is little point in learning how to use these two programs. The third project, ε -TeX follows the more general approach, and is potentially of interest to every current user of TeX.

The work done in ε -TeX is nicely thought out, and the result is both stable and virtually bug-free, but it is hardly ever used in real applications. The reason is simple: package writers will not use ε -TeX primitives until they can be certain that ε -TeX is indeed available everywhere. On the other side, ε -TeX cannot develop without input from package writers that intend to use ε -TeX. There is a chicken-egg situation, and it leads to the following conclusion:

1. ε -TeX is a nice idea with too little momentum to make a difference.

Another important problem is the fact that people that need the functionality of either ε -TeX or PdfTeX

or Ω and one of the other two extensions, cannot do so from within one document. All three have their own specific syntax extensions that are hard to fake in one of the other extensions. This is unsolvable in the current situation, and leads us to the following statement:

2. Ω , PdfTeX and ε -TeX should be merged as soon as possible.

Then there is a fourth project that has just started: the New Typesetting System (NTS).

The NTS development group hopes to increase the chances of general acceptance of NTS by guaranteeing compatibility with TeX for a number of years to come. We feel that this is a error, because most of the more fundamental issues that NTS should deal with to live up to the 'New' in it's name cannot be done without sacrificing that compatibility. Issues like grid-based typesetting and better insertion control are very likely to require a completely new algorithm, resulting in a completely new implementation. Of course it is possible to do these things in parallel, but trying to implement something new while having to be really careful not to break the old implementation unnecessarily complicates development: people that want to use TeX should stay with TeX anyway.

Therefore, we urge the NTS group to reconsider their decision to stay compatible with TeX for at least the next five years.

3. NTS will be pointless if it intends to be compatible with TeX82

The next remark we have deals with the proposed modularity of the system, which is facilitated by the use of Java:

4. NTS is a step forward *and* a step backward at the same time.

A great feature of NTS will be its extensibility. This is similar in many ways to current L^ATeX packages, albeit much more advanced. Since NTS will be

* Published in MAPS 21, Najaar 1998, pages 13-19.

written in Java, one can easily extend NTS with it's own classes. We presume there will be an easy interface to extend NTS (if not, someone will just hack the sources).

In all likelihood, this will result in precisely the same problems that current L^AT_EX has:

- Users are not aware of the packages available, and so keep asking questions like: How can I make this work in L^AT_EX?
- Furthermore, the portability of source documents (the .tex or .nts? input file) will be seriously endangered. We expect to see things like:

```
Error: this .nts style-file requires
module x.y which has not been
installed on your system.
```

The NTS team should give very strict rules for these extensions, otherwise we'll end up with another `\special-` similar situation. A central registry and a “head maintainer” are needed to keep track of extension modules in order to prevent these problems. It would be wise to turn this work into a full-fledged job under the control of (probably) TUG.

5. We need time to experiment and must not fall into the “every year a new version” trap.

An interesting common aspect in all current work is that only experience can lead to useful functional specifications. It is likely that NTS functionality will follow the same track.

This means that when we deal with the next generation T_EX programs, common users must be patient until the developers of extensions and macro packages trust the new features and can guarantee upward compatibility. It also means that it will take some years until ϵ -T_EX as well as NTS will be accepted as descendants.

We have to keep in mind Knuth completely rewrote his first T_EX!

Packaging of Distributions

Over the last 5 years T_EX has become a lot easier to install. The most important reasons for this are:

- Cd-roms have become available at large. These can easily hold a complete T_EX system. The old-fashioned piles of diskettes gave far too much trouble, and tape is only for professionals.
- Recently hard disk space has become so cheap that complete installations on hard disk are not unusual anymore.
- Installation scripts were made to shield users from tedious setup and configuration issues.

Still a number of problems remain because they are inherent to the way that T_EX systems work:

- A typical T_EX system consists of an incredible number of files (more than 31,415). No one really knows which parts are essential and which parts are not. In other words: every system is too large.
- “Everything” can be found on CTAN but only the most recent version. Old versions can be necessary to run old documents. Old CTAN dumps on cd-rom can be used to track down older versions, but we really need more professional version control.
- Maintenance is only feasible for professionals. Others are better off replacing the entire system, even though this will undoubtedly cause problems. The draw-back of ‘plug & play’ systems is that users have no idea anymore of the inner workings of the system. Is that a good thing or a bad thing?
- There is no such thing as an easy upgrade path. It's usually very hard if not impossible to simply add some files to a system and make them cooperate.
- Initial configuration can be automated, but reconfiguring is usually very hard. Any typical T_EX system contains dozens of configuration files in almost as many completely different flavours. As a rule they are scattered all over, and only an absolute expert can deal with this.

This leads to a number of conjectures:

6. The number of files in a typical T_EX system should be reduced by a factor 100.

We can achieve this by redefining the way any program finds its resources. A central database should be queried for any resource. This database should physically contain all resources. And of course it should be able to report (in any required level of detail) what's available. The database may eventually connect to CTAN (another database application) to retrieve resources not available locally.

This setup would allow for a minimal local installation to grow as necessary using Internet.

7. Configuration of a T_EX system should be centralized and automated.

If we can realize the previous issue this one will not be too hard. Programs should specify formal descriptions of the configuration details they need. These could then be generated through menus or automatically by scanning the current setup, i.e., querying the database.

8. Installation and maintenance should require far less expertise.

The database may occasionally query CTAN for any updates. The administrator would get short descriptions of these, with links to complete documentation. He/she could then select which ones should be installed. This could even be done silently (overnight) if you want an up-to-date system all the time. If necessary, programs will be signaled to reconfigure themselves.

This setup should also take care of the endless problems with non-portable DVI files. We should all be using the same resources and if we are not, the system should warn us about possible mismatches. If we decide to make TeX produce DVI files that require no virtual fonts at all (i.e., TeX reads VF's itself instead of the DVI driver) an important source of problems can be eliminated.

9. CTAN should have a complete index with descriptions of everything and cross-links to anything related to anything.

This is obvious now if we want the systems to interact. Uploads to CTAN will have to be checked more carefully: descriptions, specifications, version number, relations to other packages, dependencies on other resources, etc. must be supplied. Any item that doesn't comply to this convention should be moved out ('not supported') and deleted after a certain period.

We realize that this might cause a cultural shock in the TeX world, but we feel this is necessary to keep TeX alive & kicking in the next millenium:

10. Anarchy is what made TeX great, and it's anarchy again that will kill TeX.

Let's try to prevent this!

On-line Publication Wishlist

With the increasing growth of the internet, a whole new branch of documents has appeared: documents that are only or primarily intended for screen viewing. The used formats differ, but it is easy to see that there are some common issues involved in all of those: file download sizes, hyperlink support and ease-of-use are important points for all of these formats.

11. TeX is rather well suited to cater for those needs as it is, but some extensions are needed to make sure that TeX will stay/become in the leading position in this arena

For about 15 years TeX was only capable of producing DVI output. The limitations in both TeX

and the DVI format mainly concerned direct graphic support and color typesetting, but color printers were rare and the lack of graphics support could be worked around.

Although originally TeX was more or less supposed to handle everything itself, those 15 years of use have demonstrated that many applications, like color and graphic inserts, heavily depend on the DVI postprocessing stage. To a large extent, this is not feasible nor desired in on-line publication. On-line formats are all rather device independent themselves: otherwise people would have to publish several versions of the same document.

Theoretically, both PdfTeX the current trajectory using and DVI to PDF processing through dvips and the Acrobat Distiller can offer similar functionality, given that postprocessors are available to help out in the second case, but we can imagine both methods drifting apart, and we feel that the use of external programs to solve intrinsic problems adds a great deal of unnecessary complexity to the system.

12. On-line publishing needs primitive support

In fact, most of the conceptual extensions like hyper-referencing can be implemented using DVI and `\specials`. However, usage can be far more robust in, e.g., current PdfTeX, simply because hyper-referencing is built in, and there is no longer a need to run various programs in turn. The same goes for object reuse, fill-in forms, scripting (Java), and graphic inclusion.

But systems like PdfTeX also create new problems. Take for instance graphics inclusions: where originally TeX macros only had to bother with the dimensions of the needed box, on-line publishing backends have to include the file directly.

Although clever tricks can give acceptable results, all approaches to hyper-referencing based on current TeX interfere with either the explicit wishes of the author or the line- and paragraph-breaking mechanisms present in TeX.

13. TeX objects should be easily re-usable

When we look at object reuse, we see that this concept never surfaced in DVI (using `\specials`). This is probably due to the fact that specially screen-designed documents need these features, and it hardly matters for paper output.

From the users point of view, reuse may look rather straightforward (a sort of variant on copying boxes), but from the implementors eyes, object definitions are just another interfering kind of *whatsit*. And why is it interfering? Simply because TeX has

no particular mode which suppresses all interference. Yes, we can use a box, and we can let things happen at certain locations in the document that don't do any harm, but the situation is far from optimal.

When applied to for instance figure inclusion, reuse can quite easily be implemented in original T_EX (pure DVI, using Gilbert's DVIVIEW), the traditional DVI-dvips-Acrobat trajectory or Thanh's P_{df}T_EX. But PDF fill-in fields support demands for more.

To give you a real life example where objects are needed: in PDF one can define a check field with several appearances like on, off, mouse down, etc. Technically this means something like this (in P_{df}T_EX syntax):

```
\setbox0=\hbox{${\star $} \pdfform0
  \edef\on {\the\pdfastform}
\setbox0=\hbox{${\bullet$} \pdfform0
  \edef\off {\the\pdfastform}
\setbox0=\hbox{${\times $} \pdfform0
  \edef\down{\the\pdfastform}
```

When defining the check field, we then can refer to `\on`, `\off` and `\down`, as in the following code:

```
\pdfannot{ ... /On \on\space0 R ... }
```

Currently P_{df}T_EX only flushes forms to the output file when are accessed. (this feature is needed because we want to be able to try out things, without ending up with redundant objects, like in a macro that tries three different methods and takes the best result).

Back to the three objects, these won't end up in the file when we refer to them in the field definition above, because the field definition is handled like a `\special`: P_{df}T_EX just passes the information through.

Therefore, we end up with invalid references: the object is referred to, but never passed to the file. What do we learn from this:

14. T_EX needs a real object model.

One with immediate as well as deferred definitions, that do not interfere with the internal lists that T_EX builds and that permits forward *and* backward referencing.

Another typicality that surfaces often in on-line documents is the fact that screen layouts tend to use a lot more page decorations and colors than traditional typesetting. This is an area where a lot of disagreement is possible, but in the real world there are lots of practical applications of this.

At TUG97 there were several presentations on graphics. The related discussions invoked a BOF session on graphic primitives. Direct inclusion of

METAPOST output (in P_{df}T_EX) had already proven that a relatively small subset of PostScript primitives could be used for advanced graphics and therefore the discussion focussed on those primitives.

These graphic primitives in T_EX are not meant for drawing free hand graphics like one would do in programs like Illustrator, Corel Draw, or indeed Freehand. Instead, they are most often (to be) used for things like visualizing statistical results, plotting functions and drawing almost-mathematical shapes that can be used to emphasize certain layouts. In these graphics, text plays a important role, and this text must preferably be typeset by T_EX. It follows that inclusion of an external file will not do, and the conclusion is:

15. T_EX needs a reliable system for in-line graphics and colors

The most important outcome of the '97 BOF session was an agreement on the way to go: define a set of extensions that permit direct METAPOST output inclusion. It was felt that this set could also suffice the needs of the mainstream graphic macro packages written in T_EX.

During the NTG 'future of T_EX meeting' the participants made the exact specification of these graphic primitives (currently to be implemented as `\specials`) one of its main goals. To this end, we had to create a formal specification of the syntax involved, and that put us right in the middle of the `\special` problems.

Our final proposal on that matter will appear somewhere else in these proceedings, but Gilbert has already done some of the groundwork. Below is his explanatory text on the `\specials` that are currently included in DVIVIEW. This text is kept here because it demonstrates very well that only a few primitive commands are enough to give almost full in-line graphics capabilities.

To allow for instance METAPOST drawings to be inlined in T_EX you need several things:

- A macro to interpret METAPOST's PostScript output. Hans Hagen wrote a set of macros for P_{df}T_EX using `\pdfliteral` commands. These macros are easy to adapt to another standard using `\special` syntax
- A primitive sub-set of PostScript commands is needed. METAPOST uses only a few PostScript commands to draw it's figures.

To actually test the inline graphics standard we needed a viewer where this support was easy to include. DVIVIEW was coming to life at that time so

it was logical to use that as a test and development environment.

All primitives are easy to interpret, except for a few things like clipping and the like. The syntax will probably change in the future when the new special syntax is standardized. Converting these specials to PostScript output (e.g., modifying `dvips`) is easy to do, since the commands hardly need any translation. Specials and stuff for inline graphics in DVIview:

```
\special{dv:startgraphic}
\special{dv:stopgraphic}
\special{dv:moveto x y}
\special{dv:lineto x y}
\special{dv:curveto x1 y1 x2 y2 x3 y3}
\special{dv:stroke}
\special{dv:setlinejoin j}
\special{dv:setlinecap c}
\special{dv:setdash offset values}
\special{dv:setlinewidth w}
\special{dv:setmiterlimit m}
\special{dv:rotate r}
\special{dv:translate x y}
\special{dv:concat x1 y1 x2 y2 x3 y3}
\special{dv:newpath}
\special{dv:closepath}
\special{dv:clip}
\special{dv:fill}
\special{dv:gsave}
\special{dv:grestore}
```

As you can see the amount of commands needed to support METAPOST output is in fact quiet small. Some explanations:

dv:startgraphic Starts a graphics figure. It saves the current position and context of the DVI interpreter. The current location is marked as (0,0). As in PostScript positive x, y draws to the right and up.

dv:stopgraphic Stops a graphics figure and restores the context.

dv:moveto x y Moves the current position to x, y .

dv:lineto x y Draws a line to x, y . This does not actually draw the line but only remembers the coordinates. The actual drawing is performed by **stroke**.

dv:curveto x1 y1 x2 y2 x3 y3 Draws a Bézier curve starting at the current point to $(x3, y3)$. The control points are given as $(x1, y1)$ and $(x2, y2)$.

dv:stroke Performs the actual drawing using the current pen-style, color and width.

dv:setlinejoin j How lines are joined. j can be 0, 1 or 2.

dv:setlinecap c How the line-endings will look like. c can be 0 1 or 2.

dv:setdash offset vals Sets the pen-style. **vals** is any number of values and specifies how long the pen is on and how long the pen is off. **offset** can be used to specify a starting offset in the **vals** pattern.

dv:setlinewidth w Sets the thickness of the current pen.

dv:setmiterlimit m Sets the miterlimit.

dv:rotate r Modifies the current transformation matrix so that everything following this is rotated r degrees.

dv:translate x y Modifies the current transformation matrix so everything following this is translated (x, y) .

dv:concat x1 y1 x2 y2 x3 y3 Multiplies the current transformation matrix with the given values.

dv:newpath Discards any present paths and start a new path.

dv:closepath Closes the current path. After this you can use fill to fill the closed path.

dv:clip Selects the current path as the clipping path. All subsequent fills and strokes are clipped to the this path. The clipping path may contain one or more closed paths.

dv:fill Fills the current path with the current color.

dv:gsave Saves the graphics state.

dv:grestore Restores the graphics state.

dv:setrgbcolor r g b Sets the current color. $r, g,$ and b are specified from 0 to 1.

dv:setcmykcolor c m y k Sets the current color.

dv:setgray g Sets the current gray-level. 0 means black, and 1 means white.

Though it is easy to extend this set and include much more PostScript operators, this is not the intention. It should be noted that complex graphics which require the full PostScript set of commands should be done by including the EPS file and let PostScript do the work.

Language extension wishlist

Removal of limitations regarding fonts The font limitations that are inherent in the TFM format should be dropped. One fairly simple way to achieve this is to make T_EX read `.p1` or `.vp1` files instead of TFMs, but it is also possible to adopt a new format

like Ω 's OFM files or even create a completely new specification.

An overview of limitations in current T_EX shows limits in almost all places: the amount of characters present in a TFM, The number of separate width / height / depth / italics-corr values, the number of ligatures and kerning pairs, math sizing stuff, etc. Almost all of these limitations are not really needed anymore; most of them were born out of Knuth's desire to use as small an amount of memory as possible.

In particular, the current implementation of math mode places some really weird demands on used fonts (some characters get really weird placement in the glyph container, e.g., integrals and delimiters are all below the baseline, and the height of the `\sqrt` sign is used to decide the width of the extension bar). This should be fixed so that it becomes possible to use non-METAFONT math fonts in a reliable way, and to facilitate the creation of new math font sets. The current situation makes it impossible to use non-T_EX math fonts from, e.g., Mathematica without *lots* of vf trickery.

These things are all very easy to fix in the executable, but it won't do any good at the moment, because we are still stuck with the TFM format.

- 16.** The way TFM and VF formats are defined and implemented is the primary cause of the current font chaos

If we want to adopt a new format, the extensibility of the syntax of PL files is to our advantage, even allowing new features to be added in the future while remaining backward-compatible. But, although there no longer is a real reason for binary file input as speed or disk space optimization, binary files *do* have the advantage of being non-editable (meaning that the chances of a user accidentally breaking them is very small).

- 17.** We need symbolic names for characters

T_EX currently uses encoding instead of glyph names. Encoding is old-fashioned and merely a speed optimizing thing. The coupling of glyph-name-character should be a T_EX internal operation.

The used named characters from the fonts should be deductible from the output (DVI) file, to prevent reencoding issues in postprocessing applications. To reach this goal, it is very likely that T_EX needs an internal naming scheme for glyphs that does not depend on font encoding. Work in this area is already being done by the ϵ -T_EX team. It is considered unlikely that using UNICODE will solve the problem, but it might well be that a solution

based on the predefined set of unicode names (the road taken by Ω) is the right way to go.

- 18.** Ligatures and kern info should be independent of the character metrics

Ligatures can be present in the current font definitions, but we would like to be able to modify the lig-table internally from within T_EX. This request has already be passed on to the ϵ -T_EX group, but it needs a more general solution than the primitives that were proposed to ϵ -T_EX (`\noligs` and `\nolig<char>`). Likewise for the kerning tables.

The mechanism by which a user loads fonts into T_EX's memory is much too simple. It should be possible to specify encodings, kerning info and ligature tables separate from the actual glyph dimensions. The ligature problem actually comprises two very different problems.

The simple case is most noticeable in typesetting verbatim stuff in non-`tt` fonts, something that is often needed for textbooks on programming languages.

The hard case comes from the fact that ligatures depend on the language, not on the used font itself. The Spanish quotation, e.g., is never needed outside of Spain, and we are all stuck with it now. Ideally, every language should have it's own ligature table, that is part of the language attributes just like `\patterns` are.

- 19.** METAFONT is becoming outdated, even if T_EX itself isn't

A new version of METAFONT is needed that can generate acceptable outline fonts instead of the now used `.pk` format, and the use of non-METAFONT fonts (PostScript, TrueType) should be simplified. As stated in a previous article, T_EX should take care of the virtuality of fonts itself. But that does not *have* to imply using `.vf` files. There are some other possible solutions that may not be as powerful as `.vf`, but are a lot less confusing: The only widely used applications of virtual fonts are reencoding and creation of composite characters.

User interface

Currently, T_EX shows a weird duality: while mostly a batch tool, there are still a number of places where user intervention is needed.

On one side, if T_EX wants to survive as a batch tool (either as a stand-alone typesetter or as backend for, e.g., SGML processing systems), it will need extensions so that it is 100% safe to run the program unattended. Thinks like breaking math formulas

and placement of figures cannot be left to TeX on its own.

On the other end of the spectrum, TeX needs a real-time graphical user interface to satisfy interactive users (maybe this can be a partial implementation, like having GUI-based equation- or table-editors). This goal can only be reached if the GUI-based tools have a foolproof TeX input format that they can rely on.

There are two probable roads we envisage:

- Moving a large number of current macros into the executable itself will avoid confusion of macro formats, but there are still problems to be solved relating to redefined primitives.
- Allowing a tokenized input in a pre-compiled format would probably be better since it circumvents these problems. The idea is that, assuming we are an external program that tries to generate TeX code, we want to be very sure that `\par` really means `\par`.

But there are some other idiosyncracies in TeX's language that needs to be dealt with as well—sometimes optional, sometimes not optional keywords and characters like equal signs; arguments with braces versus arguments that are space-delimited; confusing rules for spaces; etc.

20. In all events, the language should be cleaned up drastically.

The syntax should definately be cleaned out. Anybody who has ever tried to write a non-trivial macro will know that even if your approach in itself is correct, chances are that the macro still won't work, because of a stupid mistake with `\expandafter` or extra / too few spaces. Solutions that use markup in

the style of SGML or lisp would be vastly preferable over the current situation. The current syntax often justifies the following statement:

21. TeX's macro language encourages writing garbage

We can safely say that many sources look awful in terms of formatting, just take a look at the sources of the style used to typeset this article. (Or look at the sources of the TeXbook: the output is beautiful, the input is just ugly.) In the hands of common users, bad input becomes bad output.

22. We would profit from better programming primitives

Finally, experience shows that format files are never simple and small, like Knuth presumed they would be. Instead, format files are complex programs with numerous interactions between the various parts. TeX's macro language was never supposed to support this, and as a result has virtually no programming support. Among the missing things are data structures like lists and queue's; name spaces; control structures (like cases and while loops); signals; and reliable `\if` tests.

TeX in 2003: Part II

Proposal for a `\special` standard*

NTG TeX future working group
P.O. Box 394, 1740 AJ Schagen, The Netherlands
ntg-toekomsttex@ntg.nl
<http://www.ntg.nl>

Abstract

The text of this article is a proposal for an “endorsed” `\special` specification, to be voted on by the assembly of the TUG98 meeting. Portions of this text are reworks of an original article by Nelson Beebe, and indeed large portions of the proposal itself are also based on original work done by Beebe.

Introduction

Most existing drivers have chosen an arbitrary syntax for the `\special` strings they support. This is undesirable, for at least these reasons:

- The chosen syntax is usually unique to a particular driver, and therefore seriously compromises document portability.
- The syntax is usually not extensible in an easy way.
- The syntax cannot always be unambiguously parsed.
- The output device, or driver, to which the `\special` applies is not determinable.
- The capabilities are weak, and fail to address many of the potential uses of the `\special` command.

The `\special` syntax that we have developed, which is really an extension and modification on the work done by Nelson Beebe, resolves these objections. It has the following features:

- The `\special` string is defined to contain a program written in a small language that consists of an identification string and a command, followed by sequences of assignment statements, possibly with embedded comments.
- The `\special` language is *rigorously defined* by a programming language grammar (available on request).
- The language is *extensible*. An assignment statement consists of a keyword/value(s) pair. Several keywords are already defined, and new ones can be added without invalidating existing uses of the language.

- Keywords are typed, and constant values assigned to them must be of the correct type. The supported types are names, strings, numbers, and dimensions.
- Value string concatenation is supported in the style of ANSI C, avoiding the often severe line length limitations of text editors, operating systems, and file systems.
- Provision is made for encoding all 8-bit characters in the host character set, so that, e.g., binary printer control sequences can be incorporated as *printable*, and *portable*, text in TeX documents.
- A particular keyword, `language`, is provided to permit the user to specify the output device language, or the DVI driver, to which the `\special` command is directed.

By suitable abstractions, it is possible to create a recursive-descent parser for the language in which commands, keywords and value storage locations are provided in a table passed to the parser. The parser code is therefore completely portable, and independent of the commands and keywords in the language it parses.

We will write a table-driven parser that will accept all the commands and keywords we have defined, and this parser, written in the C language, will be included in the DVIview program that will serve as a reference implementation. The parser itself will be available in the public domain soon, and patches will be made to at least `dvips` and `xdvi` to support this proposed standard.

A proposed syntax for the `\special` command

What does the language look like? Some examples will give the general flavor before we describe the

* Published in MAPS 21, Najaar 1998, pages 20-27.

details of the grammar. Here are some fragments of hypothetical T_EX input which show some of the `\special` commands:

```
% Display a picture with the upper-left
% corner at the current point
\special{**include pict.eps}
```

```
% Display a picture at its original
% absolute page position
\special{**overlay "pict.001",
        filetype metapost}
```

```
% Display a figure at half size
\special{**include "pict.eps",
        scale 0.5 0.5}
```

```
% Switch to a different colour
\special{**colour .09 .06 .6,
        model rgb}
```

Naturally, the details of a `\special` command invocation should be hidden away in suitable macros that are easy to use.

The language grammar

Nelson Beebe presented a formal grammar for his language in the article (give citation and . . .). For the purposes of the current proposal, that grammar is not repeated, we provide a textual explanation.

We will start by defining the various primitive types that are supported:

Spaces. White space is ignored except as delimiting characters, so the specification can be formatted for readability, or for compactness. Token may not contain embedded blanks (except strings of course).

Comments. Comments are from percent to end-of-line, like in T_EX. Comments cannot occur inside of strings or keywords, so this is not a comment:

```
\special{**message "Here % is some text"}
```

and this is in fact illegal:

```
\special{**mes% neat eh?
        sage "Here % is some text"}
```

Names. The grammar states that an **extended letter** is a digit, letter, hyphen, dot, or underscore. These are the characters that are allowed in commands, keywords, alternative values and unquoted strings. Lettercase is not significant in these cases.

The characters permitted are chosen such that, for instance, simple filenames can be used without surrounding quotes (see below for more info on strings and alternative values).

An “alternative value” is actually a string with some predefined values.

Numbers. Numeric constants are parsed by the ANSI C library routine, `strtod()`, which expects to see numbers in the form:

```
[whitespace][sign][digits][. digits]
[{e|E}][sign]digits
```

Dimensions. Dimensions can be given in any absolute unit known to T_EX (`bp cc cm dd in mm pc pt sp`). Note that the font-specific `em` and `ex` are not allowed. Since tokens may not contain embedded blanks, 210mm is legal input, but 210_{mm} is not.

Any keyword that accepts dimensions as arguments will also accept numbers. In the absence of a dimensional unit, a default value will be used. This default can be defined with a separate `\special` (see below under `defaultunits` for important usage information), or, in the absence of that `\special`, the driver will presume scaled points (`sp`).

Strings. The grammar supports unquoted strings and two kinds of quoted strings.

An unquoted string has to be one word only (since there are no spaces allowed), and can only use the characters that are legal **extended letters** as defined above.

The *normal* kind of quoted string is delimited by double quotes, and inside it are recognized all the escape sequences supported by the C language. The *raw* kind is delimited by single quotes; only escape-single-quote pairs are recognized inside it. This is more convenient when it is necessary to have strings with several backslashes, since it then avoids having to double all of them. Once normal and raw strings are parsed, they are stored identically.

Backslashes in literal strings and filenames pose a small problem for the user, because T_EX will ordinarily try to interpret control sequences triggered by backslashes in the argument of the `\special` command. Although it would have been possible to choose another escape character than backslash for such strings, this would likely prove confusing to those users who are used to C and UNIX, where the backslash escape character is firmly entrenched.

Fortunately, the solution is not difficult, because T_EX does not have backslash hardcoded as a control sequence prefix; you can change it by altering T_EX’s catcodes.

In the descriptions of the `\specials` below, the characters `n` and `m` are used to indicate a value from a fixed set of alternatives, `s` is used to indicate all sorts

of strings, **x**, **y** and **z** (possibly with numeric tags) are used for dimensions, and **a** through **j** are used for numbers.

Now let us move to the portions of a `\special` that actually define things. The structure of a `\special` command is as follows:

ID bytes. The first 2 characters in every `\special` are to be the two tokens `**`. The rationale behind this is that a convention like this makes it easier to adjust programs that have to remain backward-compatible with their old private syntax. As far as we know, this particular sequence of tokens is never used in current `\specials`.

Command. The next word is the principal command for this `\special`. Depending on the command itself, it may have arguments or it may be a single command.

Assignments. Optionally, the command can be followed by a series of keywords that supply extra information. Keywords follow the same syntax as commands, so there can be zero or more arguments to a keyword.

In a series of assignment statements, the order of the keywords is not significant, except that if duplicate keywords are specified, the value of the last one is used.

Every keyword-value group needs to be separated from the previous one by a separator, which may be either a semicolon or a comma. This is correct:

```
\special{**include "pict.eps";
          scale 0.5 0.5}
```

And this is not:

```
\special{**include "pict.eps"
          scale 0.5 0.5}
```

Separating items. Finally, the assignment statement may use either the equals or colon operator, or the operator may be omitted altogether. This supports the common forms:

```
\special{**include=pict.eps}
\special{**include:pict.eps}
\special{**include pict.eps}
```

Because the values have very limited syntactical possibilities, there is no ambiguity created by this.

The `\special` language

The preceding section defined the grammar for the `\special` language. We now need to define what commands and keywords will be recognized. As emphasized above, the language is *extensible*, and the parser that we will implement for it makes it easy to

add new commands and keywords *without touching a single line of the parser code itself*.

However, we presume that there will be a maintainer or maintenance group assigned to take care of this specification, and this person has the right to refuse to accept extensions that do not fit in.

Generic keywords. The full set of commands and keywords that are recognized is given below, but we will start off with some general keywords. These keywords can be used within any `\special`, and also be used as a command. They will not be mentioned separately in the descriptions of the other `\specials`:

Keyword	Value	Action
<code>message</code>	<code>s</code>	Supply an operator message to be sent to the terminal and log file.
<code>id</code>	<code>n</code>	Supplies a name that uniquely identifies this <code>\special</code> .
<code>use</code>	<code>n</code>	Supplies a name that identifies a previously defined <code>\special</code> .

The `message` string provides a means for operator communication; for example,

```
message "Thesis bond paper for this job"
```

The message is sent verbatim to the terminal and the log file.

The `id` allows identification of the `\special` it occurs in. The command and the keywords and values associated with this `\special`, are saved and available for later reuse through `use`. The current location in the file is also saved, for later retrieval by one of the cross-link `\specials`.

The usage of `use` is as follows: first, all of the data from the `\special` to which it refers, minus the `id` value, are inserted in the current `\special`, and any other values that occur in the current `\special` are used to override the inherited options. An example is probably the best way to show this. After

```
\special{**include "pic1.eps";
          scale 0.5 0.5;
          id mypic}
```

The following command re-does precisely the same in a later portion of the document,

```
\special{**use mypic}
and
\special{**use mypic;
          scale 1 1;
```

```
id mypic2}
```

inserts the same figure, but at a different scale. It also assigns a new `id` to this current `\special`. The following is also allowed

```
\special{**include "pic2.eps";
  use mypic;
  id mypic2}
```

but it is *not* legal to switch to an entirely different command, like `overlay`.

Drivers are allowed to set an upper limit to the number of *distinct* `ids` that can be used in a document, but this limit should not be lower than 256. There is never a point to limit the total amount of `ids`, since later definitions will just overwrite the previous one with the same name.

There is at the moment exactly one command that affects the `\special` parser itself:

Keyword	Value	Action
<code>defaultunits</code>	<code>n</code>	Sets the default units to one of the defined dimension types instead of <code>sp</code> .

Commands for graphics inclusion. There are three possible ways of including a graphic figure file from disc:

Keyword	Value	Action
<code>include</code>	<code>s</code>	Insert file contents with relative page positioning.
<code>overlay</code>	<code>s</code>	Insert file contents with absolute page positioning.
<code>underlay</code>	<code>s</code>	Insert file contents with absolute page positioning.

The filename string can be used for normal local files, but it can also be used for URLs, following the normal rules for URL specification. If no explicit protocol (like `http` or `ftp`) is given, the name is assumed to be a local file. Even non-networked drivers are required to correctly handle one protocol: `file://`.

In all these three cases, drivers can opt to give a default search path for figure files with relative path names, but this is not required nor encouraged. The driver is not required to include any file type except `dvi`.

`overlay` and `underlay` are supposed to start from the lower-left corner of the physical page, with

coordinates as in PostScript: up and right are positive values for `x` and `y`. In cases where there is no obvious lower-left corner (as may be the case for on-line backends), the lower-left corner is defined to be at the end of the output medium.

`include` places the top-left corner of the image at T_EX's current point. Here coordinates are as in DVI: down and right are positive values for `x` and `y`.

The difference between `overlay` and `underlay` should be clear: overlays can actually obstruct other images and text on the page (depending on where precisely on the page the `\special` was given), underlays can never do this, but a second underlay might be on top of the previous one.

If the file cannot be opened, or for relative positioning, the bounding box cannot be determined, a warning message is issued and the `\special` command is ignored.

There is also a `\special` command available for the inclusion of literal drawing commands:

Keyword	Value	Action
<code>graphics</code>	<code>s</code>	Execute the graphics primitives in string (defined below).

The `graphics` keyword value is used to insert simple generic graphics commands in one of the existing (mini-)languages for graphics. These are properly handled by using the `graphics` and `type` keywords together.

```
\special{**graphics = "...",
  type = tpic }
```

The driver will issue an error if there is a `graphics` command without a `type` specified as well, and the corresponding `\special` will be ignored. The driver is not required to execute `graphics` unless the `type` is `dvi`.

All four graphics `\specials` accept the following options:

Keyword	Value	Action
<code>boundingbox</code>	<code>x1 y1 x2 y2</code>	Defines the four dimensions of the lower-left and upper-right corners of the box which bounds the figure.
<code>clipbox</code>	<code>x1 y1 x2 y2</code>	If present, clipping to the specified four dimensions should occur.

position	n m	Specify the reference point on an inserted figure which is to be mapped to the current page position.	translate	x y	Defines two dimensions that shift the figure's reference point from the default value.
size	x y z	three values that are absolute dimensions for the size of the figure.	scale	a b	one or two numbers that are relative to the 'normal' size of the figure.
type	s	gives a way to specify the type for files with non-standard extensions.	rotate	a	rotation angle in degrees. Counterclockwise is positive.

boundingbox also applies to **graphics**, since it can be used to decide whether and where clipping should occur. Note that this is essentially the same value as the PostScript BoundingBox for (E)PS figures. For clipping purposes, this statement overrules the in-file version of such a BoundingBox. In the absence of a **boundingbox** keyword, (E)PS and similar file formats where it is legal to draw outside the box should *always* be clipped to the in-file values.

The **position** keyword specifies two values. The first should be one of **top**, **middle**, or **bottom**, and the second should be one of **left**, **center**, or **right**. These words may be abbreviated to a single letter if desired. Together, they select on the bounding box one of nine points (four corners, four edge centers, and the box center) which is to be placed at the T_EX current point. If this keyword is not given, the default is

```
position = top left
```

The point selected by this keyword (or by default) will be the *reference point* for the insertion of the graphic file.

In the values of **size**, a negative dimension means that size in that direction should be ignored.

The string argument to **type** is used to give information about the type of file or **graphics**. This value should be either the 'normal' three-letter extension for this type of file or the name of a graphics description language. The following language names are predefined: **dv**, **dvi** (ordinary binary dvi commands), **epic**, **encapsulated postscript** (also **eps**), **eepic**, **emtex**, **fig**, **metapost** (also **mp**), **pcl**, **pdf**, **postscript** (also **ps**), **tektronics**, **tpic**, **xpic**.

Generic graphics keywords. There are three keywords that define transformations. Actually these belong to the graphics language, but they can also appear inside figure **\specials**, which is why they are explained here.

Keyword	Value	Action
----------------	--------------	---------------

These three keywords can be used as stand-alone commands, in which case they apply until explicitly stopped by means of one of the commands we will define below, or they can be included inside one of the four **\specials** for figure inclusion, in which case they only apply to the subject of that **\special**.

The keyword **size** is processed before taking any transformation commands within the same **\special** into account.

Rotations, etc., that were in force at the time the figure **\special** was encountered, *are* taken into account before the calculations for inclusions are done. Here is a small example that demonstrates possible usage:

```
\special{**gsave}
\special{**scale=2 2}
    Some large text here
\special{**rotate=45}
    Large and rotated text
\special{**include test.eps,
        rotate = 45}
    This figure is rotated 90 deg CCW
    and twice as large.
\special{**grestore}
    Back to normal
```

Command for colour specifications. There is only one command defined for colour specification (well, actually two, since the American spelling "color" is also accepted), and one optional keyword:

Keyword	Value	Action
colo(u)r	?	The value should be the numbers or tokens that specify the color in the defined colour model.
model	s	The value should be a recognizable color model name.

Every driver is required to recognize the following six named values for the option string of **model**.

These are the ones that define the four most commonly used colour models: `rgb`, `cmypk`, `gray`, (also known as `grey`) and `mono` (`bitmap`).

For all these predefined colour models, a colour is defined as one or more real numbers between 0 and 1. In the absence of a `model` keyword, drivers should take the following guess as default action: if there is one number in `colour`'s value, the colour model is `grey`. If there are three numbers, the model is `rgb`, and if four, the model is `cmypk`. All other non-qualified values signify a syntax error.

Commands for the in-line graphics language.

First there are the commands that change the state of the graphics system's default values:

Keyword	Value	Action
<code>setlinejoin</code>	<code>n</code>	Select method of joining lines.
<code>setlinecap</code>	<code>n</code>	Selects the line ending method. One of <code>butt</code> , <code>round</code> , <code>square</code> .
<code>setdash</code>	offset values	Select the dashing pattern for drawing lines.
<code>setlinewidth</code>	<code>x</code>	Selects the line-width.
<code>setmiterlimit</code>	<code>a</code>	Sets the miter limit for drawing.
<code>setoverprint</code>	<code>n</code>	Value is <code>yes</code> or <code>no</code> .
<code>setvisible</code>	<code>n</code>	Value is <code>yes</code> or <code>no</code> .

Note that the commands `scale`, `translate`, `rotate` and `colour` also belong to this category.

`setvisible` and `setoverprint` are supposed to compensate for overlays and underlays as well as for the background colour of the page (defined below in the section on paper settings).

Then there are commands that draw stuff:

Keyword	Value	Action
<code>moveto</code>	<code>x y</code>	Moves the cursor position to <code>(x,y)</code> .
<code>lineto</code>	<code>x y</code>	Draws a line to <code>(x,y)</code> .
<code>curveto</code>	<code>x1 y1 x2 y2 x3 y3</code>	Draws a Bézier curve where <code>(x1,y1)</code> and <code>(x2,y2)</code> are the control points and <code>(x3,y3)</code> is the end-point.

All three commands draw relative to the current point, and in fact, they even move the driver's idea of 'current point' just like the regular DVI com-

mands do. If this side-effect is undesirable, the commands should be part of an explicit drawing, which is defined and drawn with one of the following commands:

Keyword	Value	Action
<code>startgraphic</code>		Indicates the beginning of a graphic.
<code>stopgraphic</code>		Analogously ends a graphic.

Inside one of those explicit figures, the drawing commands do not actually draw anything. Instead, one of the following commands should be used:

Keyword	Value	Action
<code>newpath</code>		Discards any present paths and start a new one.
<code>closepath</code>		Closes the current path.
<code>stroke</code>		Draws all the lines with the current selected pen.
<code>clip</code>		Selects the current path as the clipping path.
<code>fill</code>		Fills the current path with the current selected color.

Of course you are allowed to use the other commands too, and there might be intermixed text. Page breaks are not allowed though, since the entire graphics state will be restored to its default state at the beginning of the page. Usage of these commands is analogous to PostScript.

Alternatively, the graphics state can be saved and restored explicitly, again as in PostScript:

Keyword	Value	Action
<code>gsave</code>		Saves the graphics state. Position, current color, current path, current clipping path, current transformation matrix, and the current pen-type is saved.
<code>grestore</code>		Restores the graphics state.

Commands for hyper-referencing. There are not that many keywords explicitly involved with hyperlinks, since they can use the keyword `id` to mark either pages or locations within the document. The

link specification decides whether the specific `id` indicates a location marker or a page marker.

Linking re-uses the option keywords `position`, `size`, `filename` and `type` that are defined elsewhere in this paper.

Keyword	Value	Action
<code>linktopage</code>	n	The name has to be defined though an <code>id</code> elsewhere.
<code>linktoloc</code>	n	The name has to be defined though an <code>id</code> elsewhere.
<code>linkend</code>		Ends an HTML style link.
<code>position</code>	n m	Specify the reference point of the link area.
<code>size</code>	x y z	Three dimensions that are width, height and depth of the link area.
<code>filename</code>	s	This is the URL for the case of an external file link.
<code>type</code>	s	Gives a way to specify the type for files with non-standard extensions. The value should be the ‘normal’ three-letter extension for this type (like <code>pdf</code> or <code>dvi</code>).

The value of `size`, if available, gives the borders of the ‘clickable area’. An example:

```
\special{**id=1}This is a
\special{**linktoloc=1,
size=16pt 6pt 1pt}link.
```

If `size` is not explicitly given, `linkto...` functions analogous to the HTML style syntax, and `linkend` is used to stop the area. Here is an example of the this approach:

```
\special{**id=1}This is a
\special{**linktoloc 1}link%
\special{**linkend}.
```

It is a syntax error to end a link with `linkend` if that link was started with an explicit `size`, and the entire link specification will be ignored by the driver.

It is *not* an error if there is a line or even line break in the case that is supposed to end with `linkend`. These cases have to be handled correctly by the driver (the clickable area will probably have to be split into separate parts).

Commands for meta-information. A number of keywords is available to pass information to the processing application. This information can be used to fill `<meta>` tags or for debugging purposes.

Keyword	Value	Action
<code>info</code>	n	Value can be either <code>meta</code> , <code>debug</code> , or <code>comment</code> .
<code>title</code>	s	Name of the current document.
<code>subject</code>	s	Subject of the document.
<code>author</code>	s	The (probably human) author.
<code>creator</code>	s	The generating program.
<code>version</code>	s	Version information.
<code>keywords</code>	s	Keywords for this document.
<code>abstract</code>	s	Short abstract for this document.
<code>filename</code>	s	Original filename.
<code>lineno</code>	a	Records original line number in source.
<code>charno</code>	a	Records character location in line.
<code>byteno</code>	a	Records location in file.
<code>date</code>	a b c	Date in a fixed format (<code>dd mm yyyy</code>).
<code>time</code>	a b	Generation time in fixed (<code>hh mm</code>) format, assumed to be GMT.

The meanings should be clear from the names. These commands can all be used inside of any other `\special` in this same group, and they can be used in the optional part of the three figure file inclusion `\specials` and as part of the `linktoloc` and `linktopage` commands if they refer to an external file, where they can be used to request a specific version of a file. (The driver does not have to honour these latter cases in order to comply, but it is required to give the usual warning about failing to process the `\special` entirely).

Handling paper. Device initialization can be a complicated business, so it will usually require the `language` keyword as well (see below), but some of the more common keywords can be defined without problems. Paper is fairly simple. There are two commands available, `paper` and `screen`.

Keyword	Value	Action
<code>paper</code>	s	Paper form name.
<code>screen</code>	s	Screen form name.
<code>height</code>	x	Paper or screen height.
<code>width</code>	y	Paper or screen width.
<code>colo(u)r</code>	?	The value should be the numbers or tokens that

specify the color in the defined colour model.

The `paper` and `screen` keywords define a name that is used to tag the collected parameters. If the form name already exists, assignments will replace previous values. Otherwise, a new form is created. `screen` is intended for on-line formats, and is a synonym for `paper` that feels more natural in this case.

The use of `colour` here defines the background colour of the paper or screen. Printer drivers (or any other driver where execution of this command might lead to very expensive output) are supposed to ask confirmation from the user before executing this `\special`.

Other processing options. are

Keyword	Value	Action
<code>imaging_type</code>	n	The type of imaging that is applied.
<code>resolution</code>	x	Gives the required resolution for device where there are more possible resolutions.
<code>tray</code>	n	Tray number for devices with more then one input tray.
<code>duplex</code>	n	Either on or off.

The `imaging_type` can be one of the words `normal`, `negative`, `mirror` or `mirrornegative`.

The commands are used for for instance type-setter output, and they always apply to at least one full page (the page the `\special` appeared one)

Other device options. Certain drivers might require certain extra commands that only they understand. There is one command reserved to handle these things.

Keyword	Value	Action
<code>language</code>	n	Name the output-device language for which this <code>\special</code> is intended.
<code>literal</code>	s	Insert literal output device code.
<code>options</code>	s	Insert driver option.

The `language` keyword determines whether the DVI driver will process this `\special`, or ignore it.

Drivers are not required to understand *any* kind of `language` special, and are free to ignore that

`\special` right after it has seen the `language` command. However, any driver that is willing to support this `\special`, even in a very minor way, *must* recognize a generic language choice relevant to its output device, such as `PostScript` or `Epson`. Also, each driver that tries to handle this `\special` *must* recognize its own name as a language value.

`literal` is allowed to occur *only* in combination with `language`, and is used to insert literal portions of the command language used by the `language` in question.

The `options` keyword can be used to supply device-dependent information to the driver; this is only allowed if the `language` is the name of a driver.

Correct driver behaviour

Drivers are supposed to correctly interpret and execute all of the `\specials` defined in this document, except were we specifically indicated that this is not needed.

If the program that processes the DVI file *does not* know how to handle a specified `\special` (other than those defined in this document), it is allowed to issue, *at most*, one warning to the user per unrecognized `\special` type.

Since there is a reasonable chance that this DVI file at hand should have been processed by another program altogether, one warning seems prudent, but that should be enough. This rule prevents the appearance of miriads of “unknown special” warnings in documents that have parallel `\specials` for various drivers.

If the program that processes the DVI file *does* know how to handle a certain `\special`, it is allowed to issue messages, warnings or errors as it sees fit. It is always *required* to give warnings in the case of a `\special` that can only be partly obeyed. It is also *required* to give the user errors for all `\specials` that have syntax errors (assuming the driver is aware of the right syntax, which may not always be the case, but is definitely the case for the `\specials` defined here).

Calendar

1998

- Oct 1–2 DANTE, 19th meeting, Katholische Universität Eichstätt, Germany. For information, visit <http://www.dante.de/dante/DANTE19/>.
- Oct 7–12 Frankfurt Book Fair, Frankfurt, Germany. For information, contact press@book-fair.com or visit <http://www.frankfurt-book-fair.com/>.
- Oct 22 NTG, 22nd meeting, Leuven, Belgium. Topic: Fonts. For information, visit <http://www.ntg.nl/bijeen/bijeen22.html>.
- Oct 30 – Nov 1 TypeCon '98, Society of Typographic Aficionados, Westborough, Massachusetts. Principal speaker: Matthew Carter. For information, contact Bob Colby (sota@tjup.truman.edu) or visit <http://tjup.truman.edu/sota>.
- Oct 30 – Dec 1999 ABECEDARIUM: A traveling exhibition of contemporary North American work in binding, letterpress printing, calligraphy and artists' books, depicting the work of Guild of Book Workers members. First stop: Greensboro, NC. Sites and dates are listed at <http://palimpsest.stanford.edu/byorg/gbw>.

1999

- Feb/Mar DANTE'99, 20th meeting, "10 years of DANTE e.V.", Ruprecht-Karls-Universität Heidelberg, Germany. For information, visit <http://www.dante.de/dante/Tagungen.html>.
- May GUTenberg '99, Lyon, France. For information, visit <http://www.ens.fr/gut/manif/>.
- Aug 8–13 SIGGRAPH, Los Angeles, California. For information, visit <http://www.siggraph.org/s99/>.
- Aug 15–20 **TUG'99**—The 20th annual meeting of the T_EX Users Group, Vancouver, Canada. Information will be posted to <http://www.tug.org/tug99/> as plans develop.
- Sep 20–23 EuroT_EX '99, the XIth European T_EX Conference, Heidelberg, Germany. Tutorials will precede and follow the main conference. For information, visit <http://www.dante.de/eurotex99>.

For additional information on TUG-sponsored events listed above, contact the TUG office (+1 503 223-9994, fax: +1 503 223-3960, e-mail: office@tug.org). For events sponsored by other organizations, please use the contact address provided.

Status as of 1 October 1998

Hug The Lion!

Martin Schröder

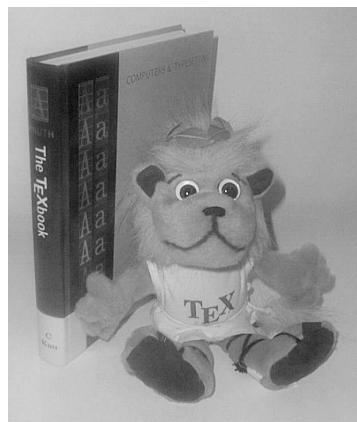


Figure 1: TEX's soft "ware" lion

Here he comes — the “Independent TEXnical Working Group on TEX Merchandising” proudly presents the one and only TEX lion as soft “ware”. This teddy lion is available for about 53 German marks from one of the following companies (among others):

- Liebscher & Partner,
Am St. Niclas Schacht 13,
D-09599 Freiberg/Sachsen,
Tel. (+49) 03731 / 78 13 86,
Fax (+49) 03731 / 78 13 77,
E-Mail info@freibergnet.de,
<http://www.freibergnet.de>
- J. F. Lehmanns Buchhandlung,
Hardenbergstrasse 11,
D-10623 Berlin,
Tel. (+49) 030 / 61 79 11-0,
Fax (+49) 030 / 61 79 11-60,
E-Mail info@lehmanns.de,
<http://www.jfl.de>

TEXies outside of Germany should contact their local TEX user groups to save handling charges by combining orders.

For every soft toy sold, 3 marks are transferred to the *TEX Merchandising Fund*. This fund serves for financing further articles to be merchandised, and encouraging TEX projects and user groups all over the world. It is administered by DANTE e.V.

◇ Martin Schröder
Crüsemannallee 3
D-28213 Bremen
Germany
Martin.Schroeder@ACM.org
<http://www.dream.kn-bremen.de>

Production Notes

Mimi Burbank

SCRI, Florida State University,
Tallahassee, FL 32306-4130
mimi@scri.fsu.edu

Well, this issue has been most interesting, and instructive—you *really* learn how to use (L^A)T_EX when you actually *do* production on a issue like this one—containing METAPOST, METAFONT, PostScript, CON_TE_XT, and pdf_TE_X! Articles were received from the TUG’98 editors by ftp, along with necessary format and font files for production of special articles.

Working with CON_TE_XT has been particularly challenging, and time was spent in understanding/translating Dutch to English, and then making changes. CON_TE_XT is a relatively powerful macro package—similar to L^AT_EX. Having speedy access to the authors is the greatest help though, and I appreciate the quick response I’ve received.

Learning how to run and update files for pdf_TE_X has been a lot of fun—more so because I also have Adobe Acrobat3.0 on my PC. After using pdf_TE_X so much with this issue, I’m convinced it is one of the most important tools that we T_EXies have today. I’m looking forward to the next installment and further development along this line.

Output The final camera copy was prepared at SCRI on IBM rs6000 workstations running AIX v4.2, using the T_EX *Live* setup (Version 3), which is based on the *Web2c* T_EX implementation version 7.2 by Karl Berry and Olaf Weber, pdf_TE_X, version 0.12h, and CON_TE_XT version 3.2. PostScript output, using outline fonts, was produced using Radical Eye Software’s dvips(k) 5.78, and printed on an HP LaserJet 4000 TN printer at 1200dpi.

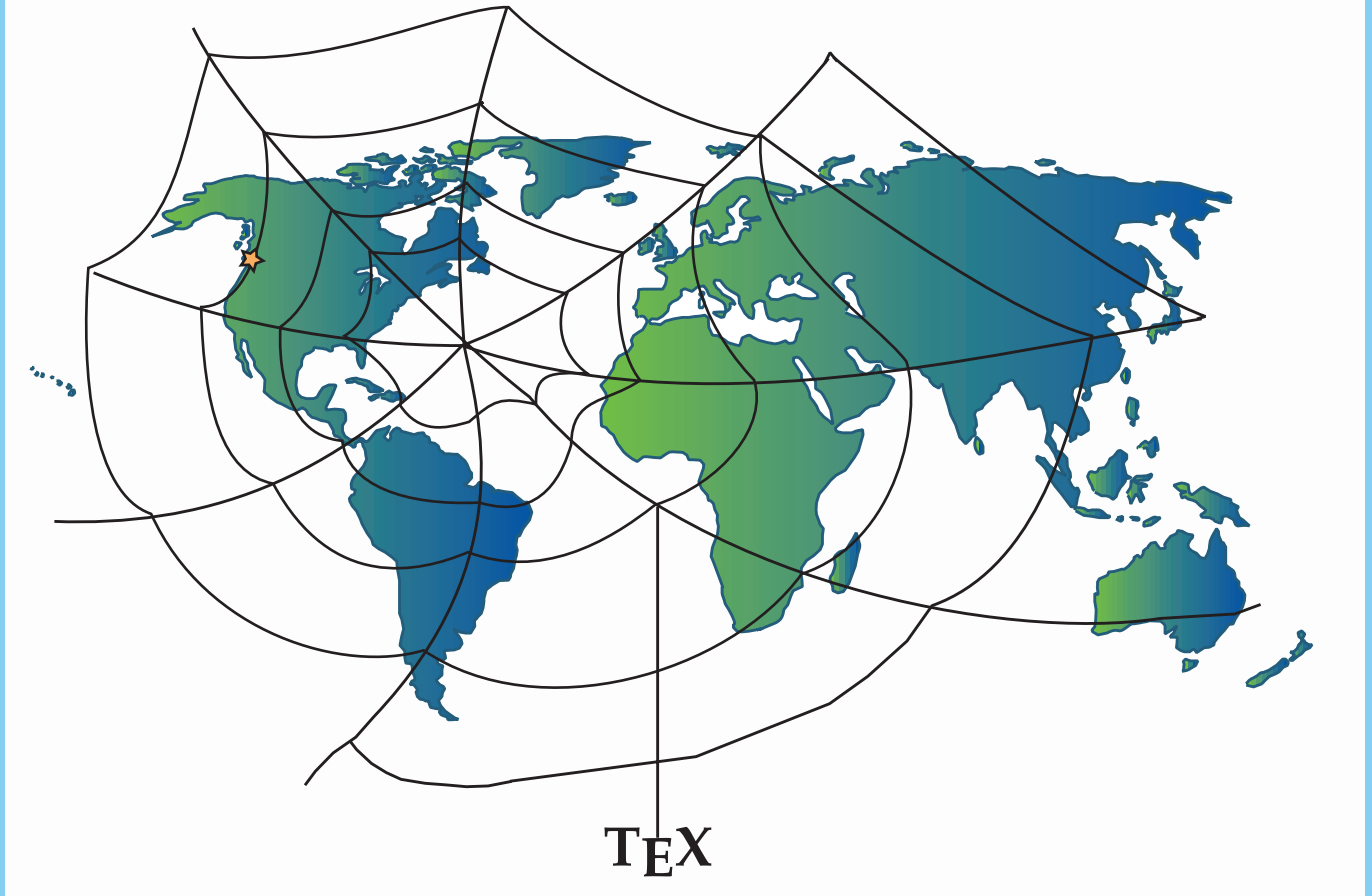
Coming In Future Issues Several articles from the conference were delayed until the December issue—Miroslava Misáková’s article about typesetting with varying letter widths; Laurence Finston’s article about typing a concordance from T_EX input files. The next issue will also have a Cahiers summary, an article by Claudio Beccari on new Greek fonts and the `greek` option of the `babel` package, an article by Anshuman Pandey entitled “Romanized Indic and L^AT_EX”, and several other goodies.

Because of the lateness of the current issue, we hope to have the December issue to you very shortly!

TUG'99 : T_EX Online—Untangling the Web and T_EX

THE 20TH ANNUAL MEETING OF
The T_EX Users Group
August 15–19, 1999

UNIVERSITY OF BRITISH COLUMBIA • VANCOUVER • BC • CANADA



PAST ANNUAL MEETINGS & CONFERENCES

1980 Stanford CA
1981 San Francisco, CA; Stanford, CA
1982 Cincinnati, OH; Stanford, CA
1983 Stanford, CA
1984 Stanford, CA
1985 Stanford, CA
1986 Medford, MA
1987 Seattle, WA
1988 Montreal, Canada
1989 Stanford, CA
1990 College Station, TX; Cork, Ireland
1991 Dedham, MA
1992 Portland, OR
1993 Birmingham, England
1994 Santa Barbara, CA
1995 St. Petersburg, FL
1996 Dubna, Russia
1997 San Francisco, CA
1998 New York City, NY; Torun, Poland

MEMBERS AROUND THE WORLD

Argentina • Australia • Austria • Belgium • Brazil • Bulgaria • Canada • Canary Islands • Chile • Costa Rica • Croatia • Czech Republic • Denmark • Egypt • Finland • France • Germany • Greece • Hong Kong • Hungary • Iceland • India • Indonesia • Ireland • Israel • Italy • Japan • Korea • Luxembourg • Macau • Malaysia • Mexico • Netherlands • New Zealand • Norway • Poland • Portugal • Romania • Saudi Arabia • Singapore • Slovakia • South Africa • Spain • Sweden • Switzerland • Taiwan • Thailand • The Netherlands • Turkey • United Arab Emirates • United Kingdom • United States • Venezuela

CALL FOR PAPERS

Abstracts Oct 17, 1998
Papers Preliminary Mar 12, 1999
Papers Preprint Jul 16, 1999

PRE-CONFERENCE WORKSHOPS

August 9–13, 1999

FOR INFORMATION CONTACT

EMAIL tug99@tug.org
URL <http://www.tug.org/tug99/>



Institutional Members

Academic Press,
San Diego, CA

American Mathematical Society,
Providence, Rhode Island

CERN, *Geneva, Switzerland*

College of William & Mary,
Department of Computer Science,
Williamsburg, Virginia

CSTUG, *Praha, Czech Republic*

Elsevier Science Publishers B.V.,
Amsterdam, The Netherlands

Florida State University,
Supercomputer Computations
Research, *Tallahassee, Florida*

Hong Kong University of
Science and Technology,
Department of Computer Science,
Hong Kong, China

Institute for Advanced Study,
Princeton, New Jersey

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

Iowa State University,
Computation Center,
Ames, Iowa

Kluwer Academic Publishers,
The Netherlands

Los Alamos National Laboratory,
University of California,
Los Alamos, New Mexico

Marquette University,
Department of Mathematics,
Statistics and Computer Science,
Milwaukee, Wisconsin

Masaryk University,
Faculty of Informatics,
Brno, Czechoslovakia

Mathematical Reviews,
American Mathematical Society,
Ann Arbor, Michigan

Max Planck Institut
für Mathematik,
Bonn, Germany

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

Princeton University,
Department of Mathematics,
Princeton, New Jersey

Space Telescope Science Institute,
Baltimore, Maryland

Springer-Verlag,
Heidelberg, Germany

Stanford University,
Computer Science Department,
Stanford, California

University of California, Irvine,
Information & Computer Science,
Irvine, California

University of Canterbury,
Computer Services Centre,
Christchurch, New Zealand

University College,
Computer Centre,
Cork, Ireland

University of Delaware,
Computing and Network Services,
Newark, Delaware

Universität Koblenz–Landau,
Fachbereich Informatik,
Koblenz, Germany

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

University of Stockholm,
Department of Mathematics,
Stockholm, Sweden

University of Texas at Austin,
Austin, Texas

Uppsala University,
Computing Science Department,
Uppsala, Sweden

T_EX Consulting & Production Services

North America

Hargreaves, Kathryn

135 Center Hill Road,
Plymouth, MA 02360-1364;
(508) 224-2367;
letters@cs.umb.edu

I write in T_EX, L^AT_EX, METAFONT, MetaPost, PostScript, HTML, Perl, Awk, C, C++, Visual C++, Java, JavaScript, and do CGI scripting. I take special care with mathematics. I also copyedit, proofread, write documentation, do spiral binding, scan images, program, hack fonts, and design letterforms, ads, newsletters, journals, proceedings and books. I'm a journeyman typographer and began typesetting and designing in 1979. I coauthored *T_EX for the Impatient* (Addison-Wesley, 1990) and some psychophysics research papers. I have an MFA in Painting/Sculpture/Graphic Arts and an MSc in Computer Science. Among numerous other things, I'm currently doing some digital type and human vision research, and am a webmaster at the Department of Engineering and Applied Sciences, Harvard University. For more information, see: <http://www.cs.umb.edu/kathryn>.

Loew, Elizabeth

President, T_EXniques, Inc.,
362 Commonwealth Avenue,
Suite 5E, Boston, MA 02115;
(617) 670-1916;
FAX: (617) 670-1916
elizabeth@texniques.com

Long-term experience with major publisher in preparing camera-ready copy or electronic disk for printer. Complete book and journal production in the areas of mathematics, physics, engineering, and biology. Services include copyediting, layout, art sizing, preparation of electronic figures; we keyboard from raw manuscript or tweak T_EX files.

Ogawa, Arthur

40453 Cherokee Oaks Drive,
Three Rivers, CA 93271-9743;
(209) 561-4585
Email: Ogawa@teleport.com

Bookbuilding services, including design, copyedit, art, and composition; color is my speciality. Custom T_EX macros and L^AT_EX₂ ϵ document classes and packages. Instruction, support, and consultation for workgroups and authors. Application development in L^AT_EX, T_EX, SGML, PostScript, Java, and BC++. Database and corporate publishing. Extensive references.

Outside North America

DocuT_EXing: T_EX Typesetting Facility

43 Ibn Kotaiba Street
Nasr City, Cairo 11471, Egypt
+20 2 4034178; Fax: +20 2 4020316
Email: main-office@DocuTeXing.com

DocuT_EXing provides high-quality T_EX and L^AT_EX typesetting services to authors, editors, and publishers. Our services extend from simple typesetting and technical illustrations to full production of electronic journals. For more information, samples, and references, please visit our web site: <http://www.DocuTeXing.com> or contact us by e-mail.

Information about these services can be obtained from:

T_EX Users Group
1466 NW Naito Parkway,
Suite 3141
Portland, OR 97209-2820,
U.S.A.
Phone: +1 503 223-9994
Fax: +1 503 223-3960
Email: office@tug.org
URL: <http://www.tug.org/consultants.html>