TEXniques

Publications for the TEX Community
Number 5

Conference Proceedings

TEX Users Group
Eighth Annual Meeting
Seattle, August 24–26, 1987

TEX Users Group
P. O. Box 9506
Providence, R. I. 02940, U.S.A.

TEX Users Group
P.O. Box 9506
Providence, RI 02940, U.S.A.

Table of Contents

Conference Proceedings for the Eighth Annual Meeting TEX Users Group

President
BART CHILDS
Texas A&M University

Vice President
RILLA THEDFORD
Intergraph Corporation

Secretary
ALAN HOENIG
John J. College, CUNY

Treasurer
SAMUEL WHIDDEN
American Mathematical Society

Executive Director
RAYMOND GOUCHER
TEX Users Group

Program Chairman, Proceedings Editor
DEAN GUENTHER
Washington State University

From the Editor

Although this is the eighth annual TUG conference, this year's proceedings mark the first time the presentations have been published. Most of the articles were typeset by the authors using TEX on their systems using their printers.

The preliminary pages for the 1987 *TEX Users Group Conference Proceedings* were set in Plantin on a Compugraphics 8600. The articles *We've Come a Long Way*, *What Should We Do*, *Driver Standards*, and *Porting TEX* were set with Computer Modern on an IBM 3820.

TEX, Linguistics was set with Computer Modern with an Imagen Designer Station. *Typesetting Greek*, *Literate Programming in C*, *Ottoman Texts*, and *Of Metafont and PostScript* were set in Computer Modern on an Apple LaserWriter or LaserWriter Plus. *Japanese TEX: jTEX* and *Driver Family* were set with Computer Modern on an Imagen 3320.

Commercial Environment was set with Computer Modern on a Cordata. *Maryland Lawyer* was set with Almost Computer Modern on an Autologic APS Micro 5. *Participants* was set with Computer Modern on an Autologic APS Micro 5.

—Ed

We've Come a Long Way, And ?

BART CHILDS

Department of Computer Science
Texas A&M University
College Station, TX 77843-3112

ABSTRACT

I will mention a few things that indicate we are beginning to succeed as an organization. I will give some detail of a few things that I think we need to address in the near future.

Those of you who participated in this annual meeting already know that this was the best meeting ever. Well that is a subjective statement, but the meeting was certainly good. This meeting and the recent issues of TUGBOAT, the other visible evidence of our organization, are indicative of my claim that we have come a long way as an organization. We also need to ask the questions of how far we should go and in which directions. I offer some of my opinions in several areas that I think are appropriate for TUG to address. These are based on phone calls that came to me because of my position as president, contacts and discussions arising from my being a site coordinator, reading the exchanges on TEXHAX, and discussions with many of you at TUG meetings. The areas are TUG membership, naming conventions and font distribution, distribution standards and source availability, and standards for drivers.

TUG Membership

Our membership has grown dramatically. I expect that it will grow even more as the technology for everybody having TEX becomes cheaper. However, reading TEXHAX shows that many (a majority ?) of TEX users either don't know about us or are too stingy to join. Can we cure this?

Fonts and Files That Contain Pixels

The use of fonts is one of TEX's most difficult items. Many users simply don't

understand which fonts are available on their system or have any documentation on how to find them. We need to make this as uniform as is reasonably possible. Some distributions use `dpi329` to indicate a directory containing **pxl** or **pk** files for `\magstephalf` fonts at 300 dots/inch. *One directory should contain all the pixel files for a given resolution and marking technology.* Thus, a directory of `pk_b300` would contain files for a **write black 300 dpi** engine. Extensions could be added for aspect ratios. Extensions on each file should reflect the magnification and format. Thus, `0pk`, `hpk`, ... `7pk` would be appropriate for the obvious zero, half, ... seven magnifications. The formats should be `pk`, `px`, or `gf`. The three character extensions will work on virtually all systems.

Standards for Distributions

TEX distributions on magnetic media that will hold 25Mbyte and above should always contain the sources that are on the standard distribution. Diskette distributions should have some form available. Since many micros don't have sufficient storage for the sources, they should probably be kept separate and charged for separately.

The standard **formats** should be included: plain TEX, LATEX, etc. Further, these should include locals like `\today`, `\time` (giving `\thetime` and `\miltime`), or a facility for including them in a standard local.

Documentation for rebuilding these formats should also be included. We should expect installations to begin using METAFONT and creating specialized fonts and logos. Each distribution should have an explicit option for **maintenance** and provisions of updates. The regularity of these updates will be dependent on extras that might be furnished.

The creation of standards for drivers will certainly cause some standard macros be furnished for TEX that will specify how graphics are to be included. These items along with the coming of color, duplex print engines, and other improvements in technology indicate that many changes will be made.

Each distribution should have a `utility` that converts pixel files from the other two standard formats to the distributions preferred. Other utilities should include importers of documents from common word processors into TEX and/or LATEX. Utilities for handling foreign keyboards are also needed to make TEX truly portable. The ß and accents like those on page 135 of the TEXbook are frequently entered as one keystroke. Some editors have the capability of converting it to the TEX control sequences, but the utility would be handy too.

Finally, we should have utilities to aid in detecting brace mismatches, removing TEX commands and checking for spelling, and easier public table macros. Some of these exist, but are not in `WEB`.

Standards for Drivers

We have an activity beginning for definition of a standard for drivers. A standard for TEX distributions will probably be harder in some sense because it has so many parts. The purpose of this is to create a start of a series of discussions that might lead to someone volunteering to lead such an effort.

Summary

I hope these issues start some discussion on what we should have in TEX distributions. I have not touched on other items like user interfaces, editor macros, and font substitution which I think are a little further into the future. More details of many of these arguments should appear in the TUGBOAT.

TEX, Linguistics, and Journal Production

CHRISTINA THIELE

Linguistics Dept.
Carleton University
Ottawa, Ontario
Canada K1S 5B6

ABSTRACT

Originally designed to set mathematics text material, and then used for other complex typesetting, TEX seems particularly well-suited to typesetting linguistics material. In addition to special characters (i.e., a phonetics font) and floating diacritics, linguistics has certain characteristic layouts: distinctive feature matrices, tree diagrams, and glossed text.

Moving beyond the specific application of TEX to linguistics, the use of a macro package in journal production also argues for using TEX; a number of advantages are discussed. The fact that authors are increasingly producing their work on computers acts together with TEX to greatly reduce production costs while maintaining the required complexity in layout and high quality in the output.

The *Canadian Journal of Linguistics* (CJL) was established in 1954 as the official journal of the Canadian Linguistic Association. Published twice a year, and containing some 40 pages per issue, the first issues were small pamphlet-sized documents. Initially printed by several different publishers, the journal finally settled in at the University of Toronto Press in 1961, where the publication remained for 23 years, until 1984. By that time, each issue was running to 80–100 pages.

In 1984, with a change in the editorial personnel, the decision was made to produce the journal at Carleton University. With the cost savings experienced

in that first year — and with a good supply of articles and reviews — the journal became a quarterly publication in 1985. Our annual number of pages doubled, from 233 in 1984 to just under 400 in 1986. The current year, only half finished, already has us at 233 pages, a pleasant coincidence. The most telling statistic is that we have doubled output while operating on roughly the same budget as in 1984. A large part of our ability to cope with reduced funding is of course due to experience in using TEX, and in making full use of computer discs supplied by authors.

The decision to produce the journal at Carleton University was prompted in large part by the arrival of a pre-release version of TEX at Carleton University. The then associate editor, JeanPierre Paillet, convinced the editor, William Cowan, that production costs could be substantially reduced if we went with TEX, without sacrificing any of the complexities associated with linguistics. With help from Rick Mallett, the site co-ordinator, TEX was put to the test.

In our early days, our roles were all quite separate and distinct, based on what we knew (and didn't know): the editor performed his editorial duties, the associate editor worked on the computer side of things (which meant all the typesetting and macro package development, undertaken concurrently), and I input the material, creating dummy codes as needed, which JeanPierre then wrote macros for.

Before beginning the first issue of CJL, we used TEX to produce another document also edited by Prof. Cowan, the *Papers of the 14th Algonquian Conference*. We were extremely proud of our efforts, a 396-page tome. TEX version 0.9 was used on the Honeywell mainframe, an old 240 dpi laser printer ran off our final copies in a building at the other end of campus, and we were pleased. No need to mention what we think of it now, in retrospect ...

With this experience in hand, we began production of our first issue of CJL for 1984 — volume 29, no. 1. It was run off oversize, photo-reduced, and printed. We were again very pleased. Our second issue of the year, however, was fraught with difficulties, as the editor and I had to become more self-sufficient. The macro package still needed some improvements, and we still didn't know how to actually "TEX" a document, much less read all the error messages we were so capably generating. We hired a student, Michael Dunleavy, to re-work the macro package, write some documentation, and to show us how to TEX our work. There was plenty of material to learn on, as we had moved from two to four issues per year. The year 1985 was our turning point as each problem was solved in turn: we became better users of *The TEXbook*, learned how to run TEX and correct our errors, and we began to think twice about typing in articles which were obviously produced on computers.

By 1986, we had most of our difficulties in hand: a macro package which gave us all the formats we required, documentation which continues to be a constant source of little nuggets of information and advice. The unsatisfactory quality of the laser printer output, and its inaccessibility had finally driven us

to purchase our own (an Imagen 300 dpi machine) in mid-1986; it is now owned by a consortium of four journals.

Our one remaining problem is our phonetics font, created by JeanPierre in bit-map form; it was designed to match only the 10pt roman font used for regular text. As well, since the change-over to the cm series, it now looks like a boldface font (the am series was heavier set). This is where a Metafont'ed version of the IPA (International Phonetic Alphabet) would be a great project for someone to undertake.[1] Much linguistic material doesn't need phonetics, but the need for decent phonetic characters in the main sizes and styles still remains.

As mentioned, there are now two other journals being produced in this fashion at Carleton, in addition to CJL and the continuing publications of the papers of the Algonquian Conference, and others are coming to see us for our typesetting. We are also looking at the possibilities of opening up some sort of publications production centre, which would provide a "group home", as it were, for the various journals, and serve as a model for others to join. The final section of this paper discusses TEX's applicability to journal production, and in particular the case of many journals using variations of the same macro package.

1. TEX and Linguistics

Linguistics text material can be distinguished from other scholarly material in two ways: the presence of special characters and symbols, including diacritics ("accents"), and the frequent use of rather complex layouts of non-paragraphed text. Special characters are predominantly phonetic symbols used to provide a one-to-one relationship between a sound and its written representation. The standard set of symbols is the IPA (International Phonetic Alphabet) which, along with regular letters, can also be modified by the use of diacritics, above or below, to the left or the right of the symbol. With respect to complex layouts, there are three main types of non-paragraph formats required: feature matrices, tree diagrams, and vertically aligned units of text, called glosses. The following sections will discuss both of these main characteristics, and the approaches used by CJL to achieve these effects.

1.1 Special Characters

TEX already comes to the user with a wide selection of special letters and symbols, which appear on the second page of chapter 9 in *The TEXbook*: æ ø œ ł ß å and so on. Some of these actually do double duty, as they are also used in the IPA (all but the last two shown here). However, the overlap with the IPA

[1] Since this summer's Annual General Meeting, Dean Guenther at Washington State University has announced the availability of a proprietary International Phonetic Alphabet. Refer to the *TEXT1* ad in TUGboat Volume 8, Number 3 or contact Dean directly.

Figure 1-a
Phonetic Font — ph10

	0	1	2	3	4	5	6	7
000	ɤ	ɗ	ʇ	ʎ	ʗ	ɓ	ɻ	ɯ
010	Φ	Ψ	Ω	α	β	γ	δ	ε
020	ζ	η	θ	ι	κ	λ	μ	ν
030	ξ	π	ρ	σ	τ	υ	φ	χ
040	ψ	ω	ɜ	ɵ	ɷ	ɪ	ɕ	ˌ
050	(	)	*	+	˛	−	.	/
060	0	1	2	3	4	5	6	7
070	8	9	:	;	¡	=	ʕ	?
100	ð	A	B	C	D	E	F	G
110	H	I	J	K	L	M	N	O
120	P	Q	R	S	T	U	V	W
130	X	Y	Z	[	“	]	ˆ	˙
140	ɨ	ɐ	ɲ	ɔ	ɖ	ə	ɾ	ŋ
150	ɦ	ɨ	ɟ	ʰ	ɭ	ɱ	ɳ	ɷ
160	ɒ	ʁ	ɽ	ʂ	ʈ	ʉ	ʌ	ʍ
170	ħ	ɥ	ʐ	ʃ	ʒ	ʑ	˗	˜

is minimal, and so a proper set of phonetic characters is required.

1.1.1 *Phonetics font:*

Clearly for a linguistics publication, a phonetics font is a priority development. The former associate editor of CJL, JeanPierre Paillet, designed pixel files for the phonetic characters, and sized them to match the regular text font used at the time. The journal was originally produced on a 240 dpi printer, giving us an over-sized font on the page, which was then photo-reduced in the hopes of making our printer's output less fuzzy. We soon stopped this and went over to printing at size: 10pt, at magstep 1, making it roughly a 12pt font. However, the phonetics font still worked well visually with the am series.

Since the font had to be hand-crafted as bit maps, we decided that we could only reasonably expect to have one set drawn, in one size, and so opted for the font to match the regular 10pt size rather than to have only a smaller sized font. Of course, it would have been ideal to have had the font in both sizes. A few characters were designed to suit a 9pt font, but in the main, where small phonetic characters occur, it's a matter of cut-and-paste by the editor — also known as the pragmatic approach to publishing.

A more recent difficulty is that the change to the cm series of fonts, which are more clean-lined, now makes the phonetics font look like a boldface whereas before it was well-suited to the am series. This means that the font is rather obtrusive where either the size is not 10pt, or the style is not roman (or boldface).

The characters shown in the font table (Figure 1-a) are the result of a number of strategies, detailed in Figure 1-b:

a. designing an entirely new character
b. modifying an existing character
c. re-orienting an existing character
d. "raiding" other font tables for characters

With respect to this last means of finding the right symbol, I have two cases which are particularly interesting.

The first is rather simple. An author required a "script v" in a footnote, in 9pt. On the basis of his hand-written model, I looked through the font tables at the back of the *TEXbook* and found the following character: ϑ. Such a straightforward "script v", and the author was quite pleased. What he didn't know was that the character was in fact from the math font's Greek characters, a variation on "theta".[2]

[2] A good example of 'form' being totally divorced from 'content', a distinction often made in traditional linguistics.

Figure 1-b

	Strategy:	Character (Code)		
a.	new:	ð (100)	ɷ (044)	ʃ (173)
		˷ (177)	˧ (176)	ʻ (054)
b.	modified:			
	non-italic Greek:	ɑ (013)	ɛ (017)	θ (022)
	nasals:	ŋ (147)	ɲ (142)	
		ɳ (156)	ɱ (155)	
	"-tail" series:	ɖ (144)	ʈ (162)	
		ʂ (163)	ʐ (170)	
c.	re-oriented:	ɥ (171)	ʌ (166)	ə (145)
		ɹ (045)	ɐ (141)	ʁ (161)
e.	from elsewhere:	<	>	~ a
		∅	→	ϑ

The other case is a very fine detail in phonetics. There are two types of the letter "a" required, each representing a different sound. One is a broken-back 'a', the other a printed 'ɑ'. The problem arises when one goes into italics: the broken-back 'a' automatically becomes a print 'a' in shape (*a*), which of course changes the sound being represented. The trick was to see italics as a type of slanted character set, so that defining the slanted font, and calling up the regular letter 'a' then gave us a passable version of an italic broken back 'a' (a and a). To add to this confusion, the unslanted TEX font cmu does the exact opposite: by typing a regular 'a', one gets a version of the printed 'a' (a and ɑ)! The chart below should sort things out:

Figure 2

broken-back			printed
roman (cmr)	a	ɑ	unslanted (cmu)
slanted (cmsl)	*a*	*ɑ*	italic (cmti)

This serves to point out that the TEX font sets are much more than just changes in angles and slopes. The font table has a potential for 128 slots to be filled, and sometimes interesting bits and pieces are used to fill in the blanks. In fact, it's a useful exercise to print out font tables of all fonts available on your system, just to see what's in some of them. For example, until you read it in the *TEXbook*, it's not obvious that an italic dollar sign will give you the pound sterling sign.[3]

[3] Which led me to ask — how does one get an italic dollar sign, and a roman

Thus the font tables can sometimes contain just the character you need, so it pays to go wandering through the rows and columns printed in the book, as well as any tables you print up yourself.

1.1.2 *Diacritics:*

Diacritics, or accents, are used to modify the sound represented by a particular letter or character. TEX comes equipped with an abundant supply of diacritics, and operates on the floating principle: they can be applied to any letter, regardless of height. Although there has been quite a bit of discussion as to how appropriately positioned some of the diacritics are (Romberger and Sundblad 1985; Levy, in this collection of papers), they do the job quite adequately. Even our own phonetics font characters can get accents, although it's a bit of an unwieldy operation: it is necessary to call up the phonetic character by its font name and \char number, enclose them in braces and then precede the whole thing with the accent.[4]. But it works: $\bar{\text{ɜ}}$ or $\acute{\text{ɔ}}$ for example.

Figure 3-a

```
\def\diatop[#1|#2]{{\setbox1=\hbox{{#1{}}}\setbox2=\hbox{{#2{}}}%
    \dimen0=\ifdim\wd1>\wd2\wd1\else\wd2\fi%
    \dimen1=\ht2\advance\dimen1by-1ex%
    \setbox1=\hbox to1\dimen0{\hss#1\hss}%
    \rlap{\raise1\dimen1\box1}%
    \hbox to1\dimen0{\hss#2\hss}}}%
```

`\diatop[\.|{\t ee}]`	e͡e
`\diatop[\‘|{\aa}]`	a̰
`\diatop[\’|{\=o}]`	ű

Double diacritics don't occur very often in linguistics material, but when they do, the typeset effect can be quite impressive.[5] As opposed to the Greek case of two marks side by side, there are times when two diacritics, one above the other, are needed, and not just in linguistics. It would perhaps seem logical to see two accents as just that: a letter with two marks above it. However, this is not the approach used in the \diatop definition (Figure 3-a), which does something else,

---

pound sign. Figure 2 gives the clue: the slanted font has an italic-like dollar sign ($\$$); the *un*slanted font has a roman-like pound sterling sign (£).

[4] See *The TEXbook*, pp. 286–287, under \accent. My thanks to JeanPierre for this reference; and for drawing my attention to the one in the next note.

[5] I don't know about the squealing mathematicians mentioned in the *TEXbook* (p. 136), but linguists can get pretty excited about it.

adding a mark to a letter already bearing a mark. That is, \diatop does not read one accent, read a second accent, and then put the two of them above a letter. Rather, it puts an accent on top of an already accented character. Combinations (which may or may not actually exist) would look like this: ǻ, or ë͎e. However, it does not always work well when one accent is to go above the letter, and another is to be below (Figure 3-b). An alternative method for such above/below double accents is to enclose one of the accents and the letter in a group, and then apply the second accent to the entire group, although again this doesn't always work:

Figure 3-b

input	output
`\diatop[\d|{\~\oe}]`	o͡e
`\d{\=o}`	o̱
`\~{\d\oe}`	~o̱e
`\diatop[\~|{\d\oe}]`	õ̱e

Another strategy might be to employ \llap, \rlap and negative and positive kerning.

The flexibility of floating accents in TEX is immediately appreciated by those setting linguistics or any other non-English text material. I've also had to reproduce some Greek lexical items — in a Spanish article — which required the math font, the math accents, and a few \llap's for the breathing marks. The presentation by Silvio Levy on preparing Greek fonts for a dictionary, including accents and breathing marks above the same letter, shows a different solution, one more appropriate to dealing with large amounts of text requiring complex combinations of diacritics and characters in Greek.

In addition to the floating diacritics inherent in TEX, we had a few interesting diacritics created. These marks are for the fine tuning of the sound represented by particular characters: ˛ ˗ ˈ ˌ (not ¸cedilla) are fairly common in linguistics material.

1.2 Formatting

Unique and interesting as some of the special characters and symbols required for linguistics are, the physical layout required for linguistics material also marks it apart from the usual humanities type of text material. In some ways, linguistics formatting is a hybrid, picking and choosing layouts from a variety of disciplines, most often mathematics. Although it might be rash to say that TEX is the only typesetting program allowing the flexibility required for linguistics material, it is not incorrect to say that linguistics is probably one of the chief beneficiaries of a

complex program like TEX, which has mainly been applied to the "STM trade": "scientific–technical–medical" typesetting.

I've chosen three layouts which represent the normal type of materials one can expect to handle in a linguistics journal. Two of them are simply applications of commands from plain tex; the third is produced using an original macro. There are as many variations in these as there are authors and linguistic theories; however, the "bread and butter" material (after paragraphs and itemised lists) is indeed phonetic features matrices (Figure 4), tree diagrams (Figures 5 and 6), and glosses (Figure 7). Generally, all these non-paragraph materials are set in a smaller font than the text, both because it looks better and is easier to read, and because large tables and diagrams can then fit into the regular-sized page. The only exception to this would be in the case of tree diagrams or matrices requiring a lot of phonetic characters, in which case the regular-sized font would be used to reduce cut-and-paste work.

1.2.1 *Distinctive feature matrices:*

Using a list of distinctive features (i.e., place and manner of articulation, tongue position, and subsidiary features), phoneticians describe sounds in terms of the presence or absence of these features. These are then presented on paper as "distinctive feature matrices", which is nothing more than a mathematics matrix, with words rather than number entries, and square braces rather than parentheses on either side.

For two-line entries, it's possible to simply plug in the prepared braces, of varying "Big-ness", of the math extension font. However, there is greater flexibility if the "grow-as-required" type of braces are used. In the beginning, feature matrices were just tables to me: a series of items listed on the page, which the editor would then enclose in hand-drawn brackets. Then I "discovered" \vcenter, and I haven't looked back. In fact, hard on the heels of having learned how to use \vcenter, two articles arrived requiring this very command; one needed 13 \vcenters, the other only three. Both were published in the same issue of CJL (Picard 1987, Walker 1987).

The description of using \vcenter (Knuth 1984:325–36) involves \hfil's and \quad's and \hbox's, and so on; and then it says "... the last example can be done much more simply [and I perk up] using the ideas of Chapter 22, if you don't mind descending to the level of TEX primitives; for example the first matrix could be replaced by ...". The chuckle is that I've been doing \haligns for three years now, and so I didn't mind one bit. It's not often that the harder route is the easier.

Since \vcenter is a standard TEX command, there's no real trick to applying it to linguistics material. The only fine details to look out for are the spacing: between the braces and the left margin of the text items, and between the top and bottom lines of text vis-à-vis the top and bottom bends in the braces. These

Figure 4

```
$$\emptyset\;\longrightarrow\left[\,%
            \vcenter{\halign{#\hfil\cr
                        \hfil C\hfil\cr
                      $-$continuant\cr
                  $\alpha$ anterior\cr
                    $\beta$ coronal\cr
      $\langle\,$+voiced$\,\rangle$\cr}}\,\right]

  \qquad\left[\,%

            \vcenter{\halign{#\hfil\cr
                        \hfil C\hfil\cr
                  $\alpha$ anterior\cr
                    $\beta$ coronal\cr
      $\langle\,$+voiced$\,\rangle$\cr}}\,\right]

  \quad\quad\left[\,%

            \vcenter{\halign{#\hfil\cr
                        \hfil C\hfil\cr
      $\langle\,$+voiced$\,\rangle$\cr}}\,\right]$$
```

$$\emptyset \longrightarrow \left[\begin{array}{c} \text{C} \\ -\text{continuant} \\ \alpha\ \text{anterior} \\ \beta\ \text{coronal} \\ \langle +\text{voiced} \rangle \end{array}\right] \Big/ \left[\begin{array}{c} \text{C} \\ \alpha\ \text{anterior} \\ \beta\ \text{coronal} \\ \langle +\text{voiced} \rangle \end{array}\right] \_\_\_\_ \left[\begin{array}{c} \text{C} \\ \langle +\text{voiced} \rangle \end{array}\right]$$

latter generally line up exactly, which makes for a slightly cramped looking matrix; it would probably be better to lengthen the braces just a touch, so that they indeed look like they are encompassing all of the text material.

The example here (Figure 4) is typical of linguistics feature matrices, showing variable features (α, β) as well as specific cases of + or − (i.e., presence or absence of a given feature). Only the diagonal and horizontal lines were added later. It's easy now to see the flaws in this example, but generally the layout is acceptable, and little different from that found in other linguistics journals such as *Linguistic Inquiry* and *Language*. In this particular case, because of the various symbols preceding the features, a \thinspace had to be inserted between the left delimiter and the start of the \vcenter (using \,); otherwise, things get a bit crowded (from Picard 1987:136).

1.2.2 *Tree diagrams:*

Tree diagrams are one of the more graphic representations of transformational generative linguistic theory. They are based on phrase structure rules, where → is read "re-written as":

$$\begin{array}{lcl} S & \rightarrow & NP\ VP \\ PP & \rightarrow & Prep\ NP \\ VP & \rightarrow & V\ (NP)\ (PP) \end{array}$$

Trees are quite formidable initially to the novice — novice linguist and novice typesetter alike. The former learns to see them as a sort of flow chart, each branching representing a further break-down in analysis of the various components of a sentence, for example. Starting with the "trunk", which represents the entire sentence, one then proceeds to its components: noun phrase or phrases, and verb phrase. Each of these can be further divided into adjective + noun, verb + adverb, and so on, down the roots, as it were, to the final match-up between syntactic component and lexical item.

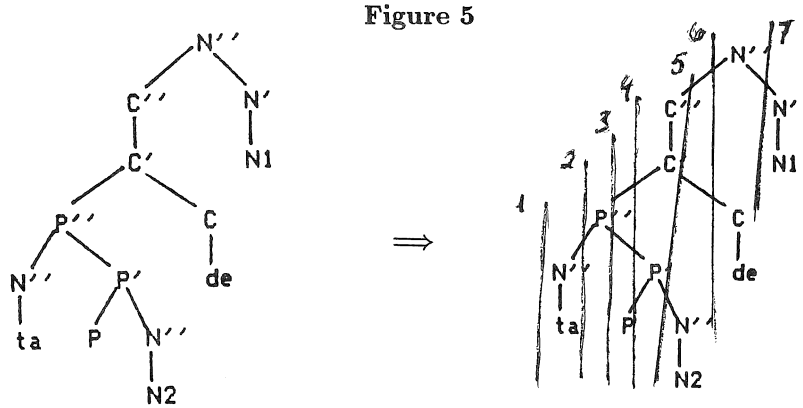
The linguist sees this is a dynamic representation. A computer whiz with a graphics program would also see it as a dynamic image. The typesetter? Well, the typesetter shouldn't be looking for where the action is. Formatting a tree requires that the typesetter *not* understand the dynamics of a tree diagram, but simply see it in terms of where the text is to sit on the page. From a pedestrian point of view, a tree is a static collection of boxes, of items suspended in a mostly white space, with lines (diagonal and vertical) connecting certain items. Boxes. The magic response to the question "how does TEX work?" — "With boxes and glue."

Once you accept the inevitability of cut-and-paste, a table layout becomes the most practical solution to trees. Think of a tree diagram as a series of rows and columns, with most of the cases empty. If the various text items in a tree are seen as contents (the # of an alignment), then the tabs represent the edges of each column.

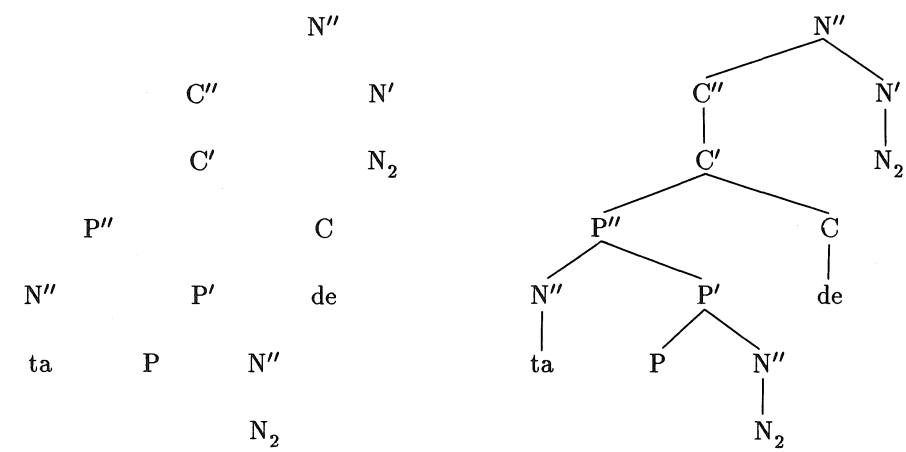
Figure 5 shows the stages involved in producing a tree diagram, using an example from a recently published article (Cheng 1986:320). It includes the original submission from the author, the approach taken to analyse the layout the author wanted (within editorial reason), the source file, and the output. Note that the output from the laser printer still required diagonal lines to be drawn — TEX could do it, but the small amount of time required to draw diagonal lines argues against spending that time on programming.

The first step is to draw in lines representing the tabs; I find it more useful to draw lines for the tab markers than for the actual contents, since all tabs (usually) must be inserted, even if some cases for contents are empty. These pencilled-in lines are then numbered, making it easier to count the correct number of tabs

Figure 5




```
\halign{\noindent#\hss\quad&\hss#\hss\quad&%
   \hss#\hss\quad&\hss#\hss\quad&\hss#\hss\quad&%
   \hss#\hss\quad&\hss#\hss\quad&\hss#\hss\cr
&&&&&&N{$^{\prime\prime}$}&\cr
\noalign{\vskip.5mm}\cr
&&&&C{$^{\prime\prime}$}&&&N{$^\prime$}\cr
\noalign{\vskip.5mm}\cr
&&&&C{$^\prime$}&&&N\lower.7ex\hbox{\tinyrmn 2}\cr
\noalign{\vskip.5mm}\cr
&&P{$^{\prime\prime}$}&&&&C&\cr
\noalign{\vskip.5mm}\cr
&N{$^{\prime\prime}$}&&&P{$^\prime$}&&de&\cr
\noalign{\vskip.5mm}\cr
&ta&&P&&N{$^{\prime\prime}$}&&\cr
\noalign{\vskip.5mm}\cr
&&&&&&N\lower.7ex\hbox{\tinyrmn 2}&&\cr
\noalign{\vfill}\cr}
```



between contents (the # in the \halign). Although it is possible to use double && in the preamble to represent repeated tab templates (*The TEXbook*, pp. 241–42), this is only appropriate when all columns in the template have the same width and quadding. In both Figures 5 and 6, it would have worked. However, there are still enough situations where each column must be individually specified as to left and right justification or centering, and distance away from the next column. The use of && is a shortcut, but may not always be the most appropriate approach.

For the preamble, I find that \halign is much more flexible than \settabs; as well, I often put the whole thing inside a $$\vbox, for centering. Usually I have a \quad between columns, and a \vskip of 1mm between lines (in a \noalign), in order to create spacing between rows. And that's all it takes. A series of & for the columns, and the little bits of text entered where they occur, until the end of the text for the tree, close off the $$\vbox, and that's it.

The output for Figure 5 has been duplicated, in order to show the difference that lines will make, rendering a seemingly random array of characters (the result of seeing a tree as a static image on paper) into a meaningful linguistic tree diagram (the dynamic representation of how a sentence is generated). The textual items are all in their correct position; lines representing the relationships between and among these elements will be inserted by hand. Tree diagrams thus become a compromise between what TEX and a user can do quickly, and how much extra time it would take to put on the finishing touches. In our case, cutting and pasting lines is more efficient.

Another example of the trees-as-tables approach is shown in Figure 6, where the idea is the same as above: lines representing a relationship between theoretical elements and their actual manifestations. This time the relationship is not between nouns, verbs and articles to a sentence, but that between hyphothetical and actual sounds in Chinese. The following example is one of over twenty such trees which appeared in an article originally submitted on a Macintosh disc (Pulleyblank 1986). Printed with the large Macintosh default font, the diagrams were very intimidating in both their complexity and size. This particular tree represents the phonetic/phonological level, and involves only two levels (p. 250). The example shows a number of interesting points, not least of which is the use of phonetic characters combined with the current cmr font.

On average, there were 15 tabs per diagram (this was before I learned the && shortcut). What these trees lacked in number of branches, they made up for in the number of columns which had to be defined, and text which then had to span several of these columns. The formatting difficulty in this case was with the use of \multispan, and usually arose when counting the number of spans required (either my math was wrong, or my judgement calls were off).

The output again shows the text items suspended in air, awaiting the addition of lines to show the historical sound changes in Chinese phonology; these have been drawn in to complete the example.

Figure 6

```
\halign{\noindent#\hss\enspace&\hss#\hss\enspace&%
   \hss#\hss\enspace&\hss#\hss\enspace&\hss#\hss\enspace&%
   \hss#\hss\enspace&\hss#\hss\enspace&\hss#\hss\enspace&%
   \hss#\hss\enspace&\hss#\hss\enspace&\hss#\hss\enspace&%
   \hss#\hss\enspace&\hss#\hss\enspace&\hss#\hss\enspace&%
   \hss#\hss\enspace&\hss#\hss\cr
\quad(c)&C&(C)&V&V&C&\lingarrow&C&C&V&C&\lingarrow&C&C&V&C\cr
\noalign{\vskip2mm}\cr
&k'&&y&a&\ng&&k'&\uglide&a&\ng&&k'&u&a&\ng\cr
\noalign{\vskip1mm}\cr
&\multispan4\hss k'y\nialpha\/\ng\hss&&&\multispan2\hss
k'\uglide\nialpha\/\ng\hss&&&&\multispan2\hss
k'w\nialpha\/\ng\hss\cr
\noalign{\vskip1mm}\cr
&&\multispan{14}\hss{\it ku\=ang}\enspace `rectify'\hss\cr}
```

(c) C (C) V V C → C C V C → C C V C

k‘ y a ŋ k‘ ɥ a ŋ k‘ u a ŋ

k‘yɑŋ k‘ɥɑŋ k‘wɑŋ

kuāng ‘rectify’

(c) C (C) V V C → C C V C → C C V C

k‘ y a ŋ k‘ ɥ a ŋ k‘ u a ŋ

k‘yɑŋ k‘ɥɑŋ k‘wɑŋ

kuāng ‘rectify’

1.2.3 *Glosses:*

A third type of layout associated with linguistics material relates to the presentation of foreign language material with translations below. Usually there are three lines of text involved: the original language, a piece-by-piece translation, and then a free-flowing English (or French) translation of the entire sentence or phrase (see Figure 7-a). The first two lines generally break the sentence up into

minimal units of meaning, called morphemes; what one would like to do is have each non-English morpheme line up with its English gloss, as shown in Figure 7-b (from Cheng 1986:322).

Figure 7-a

long $word_1$ $word_2$ $word_3$
$gloss_1$ long $gloss_2$ $gloss_3$

text of English translation

One of the original creators of our macro package, JeanPierre Paillet, designed a macro for just this layout, and then we had some variations designed subsequently, by Michael Dunleavy.

Figure 7-b

```
\def\nipar{\par\noindent\ignorespaces}%
\def\lglossit#1#2{{%
                  \setbox0=\hbox{#1\strut}%
                  \setbox1=\hbox{#2\strut}%
                  \hbox{\vtop{\box0\box1}}}}%

\def\lgloss#1#2|{\hbox{\vtop{\hbox{\ignorespaces#1\strut}%
                 \hbox{\ignorespaces#2\strut}}}\enspace}%

\def\lglosses#1@{\ifx>#1\let\next=\nipar\smallskip%
                 \else\ifx\nipar#1\nipar\let\next=\lglosses%
                 \else\lgloss#1|\let\next=\lglosses
                 \fi\fi\next}%

\def\lglossall#1<Nlglossall>{{\rightskip=0pt
plus1fil\lglosses#1@>@}}

\item{(21)}\lglossall{Wo}{I}@{pao-bu pao de}{run}@%
          {\enspace hen}{very}@{re.}{hot}<Nglossall>\nl
          'I went jogging and I am very hot.'
```

(21) Wo pao-bu pao de hen re.
I run very hot

'I went jogging and I am very hot.'

Rather than approach this as a linear, horizontal layout, the macros deal with each set of morphemes as a unit, one below the other, where the first argument (word) is the non-English piece of text, the second (gloss) is the translation of that morpheme. Inputting then becomes quite routine, with @-signs to separate each set when using the recursive versions, and no spacing decisions required of the user. Once all sets have been accounted for, there's a forced line break, and then the English translation.

The various gloss macros allow us to left-justify the entries in each set, or centre them, with some space on either side; another uses left-justification but with no space, which is also useful. And since such gloss layouts usually consist of several sets, there are recursive versions of the left-justified and the centred macros. All of these take the first item, and then position the second item below it; occasionally, one wants material above the usual line of text: we have an \overglossit to do this.

It is also possible to raise and lower the contents of either of the two arguments; this provides space to draw a connecting line between the two items, for example. One could also use an \halign for such layouts, but an \halign would approach the contents purely on the basis of shape, whereas using \glossits allows one to retain the intention and meaning of the author. The one-to-one matching of lexical item and its meaning is something which an \halign would not bring out.

2. TEX and Journal Production

I'd like to pull back from TEX and linguistics typesetting now, and take a look at the context in which they are currently used.

There are a number of points which I hope to elaborate upon in this final section. These include: the separation of the editing process from that of production; the increased use of computers by authors; the advantages to using macro packages; and how these last two can be turned to good advantage in the production of scholarly journals in the humanities.[6]

2.1 Production, not Editing

The flow chart in Figure 8 shows a generalised view of the production process we use for our journals. The first point to make, and to keep in mind, is that the process begins *after* the editor, the referees, and the author have come to mutual agreement about the contents of the article. In other words, this is a schema about PRODUCTION, not EDITING. The difference may be obvious, but with editors of small journals becoming increasingly involved in the production

[6] By "humanities", I mean both the arts and the social sciences. I use the term as opposed to "science"; others use the word as a synonym for "arts" only.

of their journals, the separation must be made explicit and retained. The use of computers by authors and production staff is appropriate, even desirable; but for a number of reasons, it is still inappropriate for the editorial work to be done with computers.

An editor may have certain hard- and software in the office, but it is highly unlikely that all submissions (once refereed and accepted) would also be produced on similar equipment, so how could an editor read and edit material on disc. Although there is a certain degree of compatibility across equipment, there are so many combinations of hard- and software possible, that the idea of using the computer for the editor's task of pushing and prodding an article into acceptable shape is simply not feasible. And even less so if one has referees involved. Note that the work of *copy* editing can and does indeed benefit from a number of computer programs which look at spelling, lexical items, even grammar. But the editing of contents, suggested changes, re-wordings and so on, all such elements introduced by an editor, referees, and an author are still best represented on paper, and most clearly understood when on paper.

In addition to these technical limitations, editorial changes in a computer file are invisible to the reader, who sees no vertical marks, no scratches, no red ink to show what was, and what now is. The potential for aggravation when changes are anonymous is clear; hardcopy with its various ink colours makes it easy to see whose changes have been introduced, or ignored.[7]

2.2 Electronic Submissions

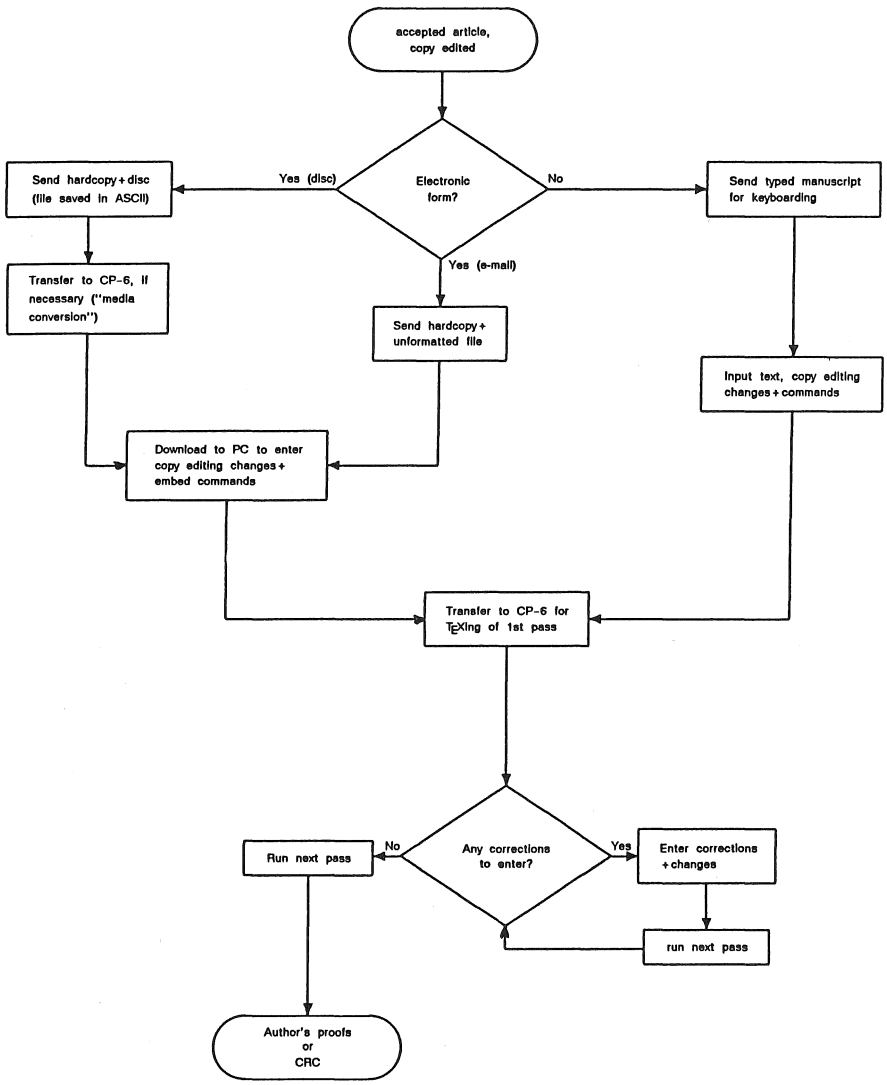
Computer use among humanities scholars is not yet common, but it is no longer the case that dedicated "hackers" are the only computer users in the humanities. In 1985, the American Council of Learned Societies did a survey of 3835 scholars in the humanities (*Scholarly Communication* 5:7), and found that:

> More than 90 percent have access to a computer, over 50 percent report that they or their research assistants routinely use a computer of some kind. Forty-five percent now own or have exclusive use of a personal computer that they use in their work. ... Wordprocessing leads the other uses; 95 percent of users rate it as very or somewhat important.

The same survey also found that "only a small minority of respondents — about 7 percent — have as yet submitted a book or article manuscript to a publisher in machine-readable form" (p. 6). Clearly the potential for electronic submissions from academics is very high, given the statistics regarding computer access; it

[7] There are two new editing programs described briefly in last year's Nov./Dec. issue of *Publish* (1986) which seem to be able to leave editor's comments and marks in the text file: *For Comment* (from Brøderbund), and *Red Pencil* (from Capsule Codeworks). I haven't been keeping up with the magazine, so I don't know any further details on these or other products.

Figure 8
Production Process for CJL



This flowchart was typeset by Lasertype, Inc., Ottawa, Canada

simply isn't being tapped. A 1984 survey (Coldeway 1984) also found low figures for electronic submissions for publication: of 340 journals listed with the MLA (Modern Language Association), 31% used computers for editing manuscripts, 19% used the machines for typesetting, but only 12% accepted discs from authors. Another 1984 survey of 150 history and 50 non-history journals (55 journals responded) found that 42% were using computers in editing, 25% for typesetting, and only 2.6% were accepting discs from authors (reported in Schnucker 1986:362–63).

In short, although computer usage is growing considerably in the humanities, the next step, to publishing, is not being followed through. Perhaps not unrelated to these low figures on electronic submissions for publishing purposes, is the apparent confusion between editorial and production uses of computers, as outlined above. Close reading of articles by editors shows that the computer is not being considered for production, but for reasons which are most often editorial in nature. Schnucker (1987:360–361) provides several examples; while his arguments against computers in the editorial process are quite valid, he doesn't take the next step following the editorial process, that is, the production phase and its potential for computer use.

The use of computers in the humanities is a fact; applying that use to publishing is what interests us here. It is this growing fact of life which, when combined with computer programs such as TEX, has begun to revolutionise small-scale scholarly publishing.

"Small-scale" refers both to the size of a journal's subscriber base (somewhere between 800 and 1200 is the norm for scholarly journals in the humanities, in Canada) and/or the size of its production staff. Especially in the case of the latter, any and all means available to reduce the amount of duplicated effort should be explored. This word "duplication" relates directly to both authors' use of computers and to TEX. That is, duplication in keyboarding, and duplication or near-duplication in layout across a variety of journals. The first case can be avoided by an increased use of submissions in electronic form. The second can be avoided by exploiting the notion of a macro package (see next section).

As far as CJL goes, we have been rather slow in taking advantage of authors' computer-generated materials, whereas the last two volumes of the *Papers of the Algonquian Conference* have almost entirely gone over to authors' discs; 68% in 1986, 79% in 1987. This has mainly been because the *Papers* contain conference papers, which the editor can solicit afterwards in electronic form. CJL is a refereed journal, with any number of articles and reviews being submitted; once accepted, one can request, but not really demand that the material be submitted electronically (although this is changing rapidly). More to the point, however, is the pragmatic reality of waiting for a disc, or typing in the materials ourselves; we often take this latter route (even though it then messes up our percentages on using authors' discs).

Nevertheless, for the sake of statistics, I've checked the files, and identified

the articles and reviews which were obviously done by computer, even if we didn't take advantage of that fact in our production. Figure 9 shows the figures for the past four years:

Figure 9

Canadian Journal of Linguistics	1984	1985	1986	1987 (till June)
no. of printed pages	233	512	399	233
% submitted in electronic form	10	43	82	74
% of electronic submissions used	0	11	53	68

The point to all of this is to underline the fact that duplication in keyboarding is one area where computers and authors' discs can substantially reduce the amount of time and money spent on basic data entry by a journal. It also quite clearly greatly reduces the number of typos, dropped lines, and misinterpretations. This time and money can then be channelled into the typesetting of articles and reviews for publication. (And I am going to concentrate on journal production, although books and other documents, such as theses, also enjoy the same benefits and advantages.)

2.3 Macro Packages

I've mentioned above that the notion of "duplication" also relates to TEX. The duplication I have in mind is that of duplicating or repeating a certain layout consistently, both within a given document, and across numerous issues of that document. This repetition of layout is one of *the* distinguishing characteristics of a journal, as compared to a book (other than a book series). A journal follows a consistent pattern in its layout, its font styles and sizes, page dimensions, etc. Once the pattern has been established, or "designed", production becomes a matter of "plugging in new values" for each issue: articles and reviews, and advertising spots if they exist. Repetition is precisely what computers are so good at, so why not apply this to journal production? If repetitious consistency is what a journal needs, then TEX and macro packages are admirably suited to the task.

I believe that a well-designed, comprehensive macro package is probably the most crucial component to journal production using TEX or any other sophisticated program, especially when it comes to producing a variety of journals. I have a list of several advantages which I've seen over the past few years:

1. Once a macro package has been designed, covering all aspects of large document production (size parameters, fonts, special formatting features, all the way to the

output routine), then there are only about 20 to 30 commands and parameters which require attention when modifying the output for a new publication.

2. Modifying these commands and parameters is anywhere from very easy to moderately difficult, since there are only so many ways to lay text on a page, to position headers and footers, and so on. That is, not only is the number of commands to modify fairly low, the amount of skill/time involved is far less than would be needed to write a whole new package. This does not, however, mean that the journals must become almost indistinguishable. The flexibility TEX affords can create very different looking pages; the similarity in commands does not lead directly to similarity in appearance.[8]
3. And of course, TEX can be used as a programming language to create new macros. Although macro writing is still a relatively new skill for me, there are any number of other users who have expressed their great satisfaction and pleasure in TEX's flexibility when it comes to writing new macros.
4. "New" journals can be input right away using existing commands, while the actual results of those codes can be worked out in the macro package. Thus, package development is done in tandem with inputting, rather than wait for finalization of commands and definitions before proceeding.
5. Users can input the same commands into articles which may be destined for different journals; the difference lies in the package specifications, not the commands. This allows for great flexibility in staff, and in training: it's not a question of knowing a specific journal's layout so much as knowing the commands for certain blocks of text, and "signature" markings (choice of headers/footers/subheading style, etc.).
6. The choice of word processing program and computer is independent of TEX. This means that personnel already skilled in the use of a particular software package can begin immediately to input TEX commands, as well as input the text, using what they know. Thus, rather than have a new person learn three things (new machine, new word processing program, new formatting commands), they only need to learn one: the formatting commands of TEX.
7. Any documentation written for the original macro package is then directly applicable to users of package A′, or A″, etc. That is, documentation describing the use of commands is still valid, even if the results of those commands may have been altered.

To summarise then, the flexibility which a TEX macro package affords, when coupled with much-reduced keyboarding due to using authors' discs, makes small-scale publishing of scholarly journals not only feasible, but economic, while retaining a good to high level of quality in the final product.[9]

[8] See Horowitz 1987 on the joys of diversity.

[9] In fact, if the output has to be top-notch, the dvi files can be shipped off to

3. References

Cheng, Lisa Lai Shen
1986 "*de* in Mandarin". *Canadian Journal of Linguistics* 31:313–326 [shown here as Figures 5 and 7].

Coldeway, John C.
1984 "A Report on Computer Use by Learned Journals". *Bulletin of the Council of Editors of Learned Journals* 3:3–9.

Horowitz, Irving Louis
1987 "Limits of Standardization in Scholarly Journals". *Scholarly Publishing* 18:125–130.

Knuth, Donald E.
1984 *The TEXbook*. Reading, Mass.: Addison-Wesley.

Morton, Herbert C., and Anne Jamieson Price
1986 "The ACLS Survey of Scholars: Views on Publications, Computers, Libraries". *Scholarly Communication*. No. 5 (Summer).

Picard, Marc
1987 "Properties of Consonant Epenthesis". *Canadian Journal of Linguistics* 32:133–142 [shown here as Figure 4].

Publish!
1986 "Electronic Editing Software". Vol. 1, No. 2 (November/December), p. 36.

Pulleyblank, Edwin G.
1986 "Some Issues in CV Phonology with Reference to the History of Chinese". *Canadian Journal of Linguistics* 31:225–226 [shown here as Figure 6].

Romberger, Staffan, and Yngve Sundblad
1985 "Adapting TEX to Languages that Use Latin Alphabetic Characters". *Proceedings of the First European Conference on TEX for Scientific Documentation*. Dario Lucarella, ed. Reading, Mass.: Addison Wesley.

Schnucker, Robert V.
1986 "The Road of Survival for Journals in the Humanities". *Scholarly Publishing* 17:355–363.

Walker, Douglas, C.
1987 "Morphological Features and Markedness in the Old French Noun Declension". *Canadian Journal of Linguistics* 32:143–197.

---

any one of a number of commercial firms with typesetting equipment capable of handling dvi files (TEXSource, Ampersand Typographers, ArborText).

Typesetting Greek

Silvio Levy

Mathematics Department
Princeton University
Princeton, NJ, 08544

ABSTRACT

We discuss the design, creation and use of a family of Greek fonts for TEX. The fonts can be used for modern or classical Greek, by themselves or in combination with the Computer Modern family of fonts. A short historical introduction is followed by a discussion of special topics, including the handling of accents and breathings, hyphenation, and the two varieties of sigma.

A Bit of History

During the first four centuries after the introduction of the printing press in Europe, the printing of Greek was hampered by the relative inadequacy of existing types, at least in comparison with the quality and variety of the best roman fonts. This was partly a consequence of the antiquity of the language and its consequent evolution in both writing and pronunciation: not only did the letterforms change over time, but they came to be adorned with a multitude of diacritical marks, a legacy of zealous scribes and grammarians anxious to preserve the pristine state of the language that lent them their prestige.

In the early days we find that the few printers that attempted to cut Greek type generally ignored all this complexity, either disregarding the diacritics or casting them separately and setting them above each line of text. Some texts were printed in lowercase only; others would make do with roman capitals like 'A' and even some lowercase roman glyphs, like 'v' for 'ν'. (See figures 1–12 in [Brit. Mus. 1927], an excellent historical survey.)

In 1495 Aldus Manutius introduced cursive greeks. They became a resounding success, as they reproduced the florid and idiosyncratic handwriting of the day, full of abbreviations and ligatures. The inordinate number of ligatures is in fact the most striking feature of such fonts: for example, Fell's "Great Primer Greek" [Morison 1967, p. 102] has sorts for many four-letter words and prefixes.

Although skilfully typeset books using cursive fonts are undeniably beautiful (see [Brit. Mus. 1927], figure 29), the use of so many ligatures was a nightmare for compositor and reader alike, since in some cases the component letters are virtually unidentifiable. Gradually the fonts were scaled down in size, but the practice of imitating handwriting remained until the end of the eighteenth century, when Bodoni, Göschen and Didot paved the way back to a more sober course.

Still problems remained. Porson's greek [Mosley 1960], first used in 1826 and destined to become the standard of the English-speaking world, continues the illogical tradition of mixing upright capitals with slanted minuscules; in particular the blending of Greek and roman text leads to poor results (figure 1). Didot's design was somewhat more felicitous, and became standard in Italy, France and Greece itself (figure 2), but the type is narrow and irregular, bearing, according to Scholderer [Brit. Mus. 1927, p. 14], the 'malign mark' of Bodoni's greeks (which were admittedly not a match for his inspired romans).

That λόγος in this phrase originally meant 'reckoning', 'calculation' is clear from Aeschines 3 § 59 *ὅταν περὶ χρημάτων ἀνηλωμένων διὰ πολλοῦ χρόνου καθεζώμεθα ἐπὶ τοὺς λογισμούς . . . ἐπειδὰν ὁ λογισμὸς συγκεφαλαιωθῇ, οὐδεὶς . . . ὅστις οὐκ ἀπέρχεται . . . ἐπινεύσας ἀληθῆ εἶναι ὅτι ἂν αὐτὸς ὁ λογισμὸς αἱρῇ.* **Dealing with this in his reply, Demosthenes says (18 § 227)** *ὥσπερ δ' ὅταν οἰόμενοι περιεῖναι χρήματά τῳ* **('that he has a balance')** *λογίζησθε, ἂν καθαιρῶσιν αἱ ψῆφοι καὶ μηδὲν περιῇ, συγχωρεῖτε* **(for the compound cf.** *ἡ καθαιροῦσα ψῆφος* **of a vote for conviction and Dion. Hal.** *Ant. Rom.* **vii. 36** *ὅτι δ' ἂν αἱ πλείους ψῆφοι καθαιρῶσι, τοῦτο ποιεῖν).* **Herodotus has** *καὶ δὴ καὶ ὁ*

Figure 1

Ἐσπέραν δέ τινα ἐτόλμησε καὶ νὰ ἐπιψαύσῃ διὰ τοῦ ἄκρου τῶν χειλέων τὸ μέτωπον τοῦ κοιμωμένου φυγοῦσα μετὰ τρόμου, ἅμα εἶδε κινούμενα τὰ βλέφαρά του. Ὁ δὲ καλὸς Φλῶρος διηγήθη τὴν ἐπιοῦσαν εἰς τοὺς συντρόφους του πῶς νυκτερινὴ ὀπτασία εἰς κεντητὸν ὑποκάμισον περιτυλιγμένη ἐπεσκέφθη αὐτὸν καθ' ὕπνους. Ἀλλ' αἱ ὀπτασίαι, τὰ ὄνειρα

Figure 2

(Incidentally, the different treatment of capitals established itself in mathematical typesetting. In English books, Greek capitals are the only math letters

that are not slanted; in French and German books they conform with the rest. The Computer Modern Greek faces are based on Porson. They are, of course, meant for math; when used for text, as in [Wonneberger 1987], the results fall short of the perfection achievable with TEX.)

It was not until the beginning of this century that really well-designed Greek types became available, including Scholderer's 'New Hellenic' type (figure 3), based on a pre-Aldine model, and the Monotype font shown in figure 4, a much improved version of Didot's design and perhaps the typeface most favored for high-quality printing in Greece during the last few decades.

ΙΩΑΝΝΟΥ Α'

Ἡ αἰώνιος ζωὴ ἐφανερώθηκε διὰ τοῦ Χριστοῦ

Ἐκεῖνο ποὺ ὑπῆρχεν ἀπὸ τὴν ἀρχήν, ἐκεῖνο ποὺ ἔχομεν ἀκούσει, ἐκεῖνο ποὺ ἔχομεν ἰδῆ μὲ τὰ μάτια μας, ἐκεῖνο ποὺ παρατηρήσαμε καὶ τὰ χέρια μας ἐψηλάφησαν, περὶ τοῦ Λόγου δηλαδὴ τῆς ζωῆς, – καὶ ἡ ζωὴ ἐφανερώθηκε καὶ ἔχομεν ἰδῆ καὶ

Figure 3

Λίγο ἀκόμα
θὰ ἰδοῦμε τὶς ἀμυγδαλιὲς ν' ἀνθίζουν
τὰ μάρμαρα νὰ λάμπουν στὸν ἥλιο
τὴ θάλασσα νὰ κυματίζει

λίγο ἀκόμα,
νὰ σηκωθοῦμε λίγο ψηλότερα.

Figure 4

Requirements

My main motivation in designing a family of Greek fonts is the preparation of a Modern Greek–English dictionary. This in itself makes the design more exacting: the fonts should not only look good individually and in combination, but also blend well with Computer Modern fonts; there should be at least three main styles, different enough that no special effort to distinguish between them is required, even in small point size; and the typing should be as painless as possible on a standard (English) keyboard. This last condition is, of course, open to interpretation, but I define it to mean that each grapheme (letter, accent, etc.) should require as few keystrokes as possible—generally one, but occasionally more, like accents in English.

In esthetic terms there are a lot of details to work out, but the foundation has been laid. I follow loosely the type shown in figure 4, which shares several features with the Computer Modern roman: sharp contrast between thicks and thins, similar letter widths, and a wealth of texture and detail. (This is not surprising since the 'modern' family of fonts goes back to Bodoni and Didot.) But it also has distinguishing features, which recapture some of the beauty of cursive writing: its strokes are more fluid, and there is no left-right symmetry (compare a roman 'o' with a greek 'ο').

Modern Greek has traditionally been typeset with the diacritic apparatus of ancient Greek, which is very rich: it contains two accents, the acute ´ and the grave `, that can go over any vowel, plus one, the circumflex ~, that can go over the vowels αηιυω; two so-called breathings ' ', one of which goes over every vowel or diphthong in initial position; the diaeresis ¨, that can go over ι and υ; and the iota subscript ,, that can go under αηω. An accent plus a breathing, or an accent plus the dieresis, can coexist; furthermore any accent, breathing or accent-breathing combination can share a vowel with an iota subscript. Here, for example, are the 24 possible varieties of lowercase α:

ἀ ἁ ὰ ᾶ ἄ ἅ ᾴ ἃ ἂ ᾳ ᾲ ᾷ ἇ ἆ ᾀ ᾁ ᾄ ᾅ ᾂ ᾃ ᾆ ᾇ

In the last several decades the tendency has been to get rid of these complications, and now the "official" system in use in Greece includes only one accent and the dieresis, without breathings or iota subscripts. Even though I work primarily with modern Greek, however, I thought it short-sighted not to include the whole apparatus, having in mind both classicists and those traditionalists who still prefer the three-accent system, as do some publishing houses.

It was clear right away that these diacritics should be implemented as ligatures, not as TEX accents, because in TEX words that include accents cannot be hyphenated. This implied that 128 font positions were not enough. Fortunately TEX and METAFONT are well equipped to handle 256-position fonts, though most device drivers are not (see the section 'Other Problems').

Following the one-keystroke-per-grapheme rule, then, one can conjure up the last alpha in the display above by typing `>~a|`, which accesses a four-character ligature. The remaining conventions for diacritics are: `'` and `` ` `` for the acute and grave, respectively; `"` for the dieresis; and `<` for the rough breathing. (If you're wondering how to produce quotation marks, or «εἰσαγωγικά», it's by typing `((` and `))`.)

The Font Layout

A further complication has to do with the letter sigma. According to Knuth [1980], the letter 's' is in a class by itself in terms of complexity of design; its Greek counterpart is similarly difficult, though for different reasons. A sigma in initial or medial position in the word is written σ, but in final position it

is written ς. Since the alternation is entirely conditioned, it seemed a pity to require different characters in the TEX file to represent the two varieties of sigma.

The first solution I tried made no assumptions on the font. It consisted in making 's' an active character, which checked the next token and printed σ if it was a letter, ς otherwise. There were two drawbacks to this procedure: the letter 's' could not be used in control sequences, and the check was expensive. (It also didn't work when the next token was a control sequence that expanded to one or more letters.)

A better idea is to save one position in the font for an invisible dummy character. All punctuation marks (and the space) are made active; they expand to the dummy end-of-word character, plus the punctuation character. An 's' by itself prints as σ, while an 's' in ligature with the dummy—which is to say, at the end of a word—prints as ς. All other letters yield themselves when in ligature with the dummy. This is still not ideal for two reasons: TEX is always "obeying spaces," and words separated by something other than punctuation or spaces (say `\par`) count as one. (But this might be the only practicable solution for the Hebrew alphabet, which has five chameleon letters.)

The solution I chose avoids the drawbacks above, at the expense of several positions in the font. Namely, the font contains each of the possible combinations σα, σβ, ..., σω, which are automatically accessed as ligatures when the word contains 's' followed by a letter.

With all the s+letter combinations and all the vowels with diacritics, it turns out that not even 256 characters are enough. Since I couldn't push the font size any further, I decided to eliminate some of the vowel + diacritic combinations. The obvious candidates were the combinations of breathings with grave accent, which can only occur in a restricted number of monosyllables, and thus can be typeset as accents, because no hyphenation is required anyway. So I made the characters < and > active. Depending on whether or not the next character is `, these active characters expand to an `\accent` or to a `\char`, the latter meant to form a ligature with what comes next. (Actually, things are not quite so simple. A breathing or accent over a capital vowel is traditionally written before the letter, so the `\accent` control sequence is only emitted if the following character is lowercase.)

The complete layout of the fonts is shown on the next page. There are a few unfilled positions, two of which I'm saving for the digamma, whose design I haven't yet tackled.

Other Problems

In order to write continuous text, I had to prepare a modern Greek hyphenation table, which I'm currently testing. The hyphenation of modern Greek follows closely that of ancient Greek, which is straightforward (that is, described by a fairly short set of rules) because Greek had originally a phonetic script—one

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
'00	–	σα			σδ	σε	σφ	σγ
'01	ση	σι	σθ	σκ	σλ	σμ	σν	σο
'02	σπ	σχ	σρ	σσ	στ	συ	σβ	σω
'03	σξ	σψ	σζ		‘	’	˘	–
'04	¨	!	¨	˛	˛	%		′
'05	(	)	*	+	,	-	.	/
'06	0	1	2	3	4	5	6	7
'07	8	9	:	·	‹	=	›	;
'10	ˇ	A	˘	˚	Δ	E	Φ	Γ
'11	H	I	Θ	K	Λ	M	N	O
'12	Π	X	P	Σ	T	ϒ	B	Ω
'13	Ξ	Ψ	Z	[	˝	]	˛	˚
'14	`	α		σ	δ	ε	φ	γ
'15	η	ι	θ	κ	λ	μ	ν	ο
'16	π	χ	ρ	ς	τ	υ	β	ω
'17	ξ	ψ	ζ	«	¸	»	˜	—
'20	ὰ	ἀ	ἀ	σὰ	ᾰ	ᾱ	ᾱ	σᾰ
'21	ά	ἄ	ἄ	σά	ᾰ́	ᾱ́	ᾱ́	σᾰ́
'22	ᾶ	ἆ	ἆ	σᾶ	ᾰ͂	ᾱ͂	ᾱ͂	σᾰ͂
'23	ὴ	ἠ	ἠ	σὴ	ὴ	ἠ	ἠ	ση
'24	ή	ἤ	ἤ	σή	ή	ἤ	ἤ	σή
'25	ῆ	ἦ	ἦ	σῆ	ῆ	ἦ	ἦ	σῆ
'26	ὼ	ὠ	ὠ	σὼ	ῳ	ῳ	ῳ	σῳ
'27	ώ	ὤ	ὤ	σώ	ῴ	ῴ	ῴ	σῴ
'30	ῶ	ὦ	ὦ	σῶ	ῷ	ῷ	ῷ	σῷ
'31	ὶ	ἰ	ἰ	σὶ	ὺ	ὐ	ὐ	σὺ
'32	ί	ἴ	ἴ	σί	ύ	ὔ	ὔ	σύ
'33	ῖ	ἶ	ἶ	σῖ	ῡ	ὖ	ὖ	σῡ
'34	ὲ	ἐ	ἐ	σὲ	ὸ	ὀ	ὀ	σὸ
'35	έ	ἔ	ἔ	σέ	ό	ὄ	ὄ	σό
'36	ΐ	ἲ	ἲ	ΐ	ϋ	ὒ	ὒ	ῦ
'37	ᾳ	ῃ	ῳ	ῤ	ῥ	σ`	σ´	σ˜

letter for each phoneme and vice versa. This conservatism means that hyphenation does not necessarily occur between syllables, as in the word σταυ-ρός 'cross', now pronounced [sta´vros]; but this apparently doesn't bother anyone.

One difficulty, however, cannot be resolved by means of mechanical rules: the digraphs μπ, ντ and γκ are sometimes pronounced as nasal + voiced stop, sometimes as voiced stop alone, depending on the word (also sometimes on the speaker). In the first case, the group can be split, but in the second, one should hyphenate before the group. This problem seems to be solvable only by trial and error.

The last problem I want to discuss is that of device drivers. We have been using various `dvi`-to-PostScript drivers in Princeton, and I found that none of them would work with 256-character fonts, even though both TEX and PostScript are designed to handle such fonts. I was, however, able to adapt Nelson Beebe's excellent driver `dvialw` [Beebe 1987] after making only three changes, because the program is well written and well documented. I exhort all those who write `dvi` drivers to include 256-character fonts in their design, since they will certainly become more necessary as TEX extends its reach around the world.

Bibliography

Nelson H. F. Beebe, Public domain TEX DVI driver family, *TUGboat*, **8**:1 (1987), 41–42.

British Museum, *Greek Printing Types*, London, British Museum, 1927.

Donald E. Knuth, The Letter S, *The Mathematical Intelligencer*, **2** (1980), 114–122.

Stanley Morison, *John Fell: the University Press and the 'Fell' Types*, Oxford, Clarendon Press, 1967.

J. M. Mosley, Porson's Greek Types, *Penrose Annual*, **54** (1960), 36–40.

Reinhard Wonneberger, Typesetting 'Normaltext', *TUGboat*, **8**:1 (1987), 63–72.

The Ottoman Texts Project

Walter Andrews and Pierre MacKay

Department of Near Eastern Languages and Civilization
University of Washington
Seattle, Washington 98195

ABSTRACT

The Turkish orthographic reform of 1928, which required the abandonment of Arabic script in favor of a Latin letter alphabet, was accompanied by a cultural rejection of all literature from the Ottoman period of Turkish history. As a result, only a small part of Ottoman Turkish literature has been made available in scholarly editions in the new orthography. The Ottoman Texts Project is a cooperative effort of Turkish and North American scholars to provide new editions of these works using popular low-priced personal computer systems and standard general purpose software. This paper describes an approach based on the adoption of TEX as the preferred output system for publication.

The Ottoman Texts editing and typesetting project represents an attempt to provide a simple, low-cost system for the entry, editing, and typesetting of transcribed [romanized] Ottoman Turkish texts. The purpose of developing such a system was to take advantage of the increasing availability of microcomputers world-wide and to induce the editors of Ottoman texts– especially Turkish editors–to employ electronic media for their editing tasks. The benefits to scholars of having a large corpus of texts available in machine-readable form seem obvious, but overcoming "technology cringe" on the part of scholars whose devotion to medieval literature stems in large part from a strong conservative-traditionalist ideological bent is no small task. Nonetheless, the rewards of converting a significant number of such scholars would be quite high. The vast

majority of significant Ottoman Turkish texts await up-to-date editing and the suggested technological change could have a major impact on the speed and accuracy of the editing process as well as on the development of lexicographical tools and on many areas of literary and linguistic study.

The situation in Ottoman studies that makes a switch to electronic media especially attractive at this time is rather complex and demands some historical introduction. From its earliest years at about the beginning of the 14th century until early in the 20th century, the Ottoman dialect of Turkish was written in the Arabic script. The political decline of the Ottoman Empire from its pinnacle of world power in the 16th century to its status as a moribund, defeated ally of Germany following World War I, was arrested in the first three decades of this century by a political and ideological revolution that saw the establishment of a Turkish Republic and an accompanying rejection of the literary, cultural, and religious institutions of the Ottoman past. One aspect of the cultural revolution was the adoption of a latin letter alphabet for Turkish, a change which had among its consequences the expansion of literacy beyond a small elite circle to the general populace, a conscious effort to simplify the written language, and a resultant major decline in the ability to read and comprehend the Ottoman literary language in any of its forms. The ethos of the early years of the Republic, to which the Ottoman Empire appeared as decadent and its culture as derivative, also meant that, at a time when the scholarly edition of older texts was becoming a growing concern in other parts of the world, in Turkey interest in things Ottoman, including Ottoman texts, was considered backward, anti-nationalist, counter-revolutionary, and wrong-headed. As a result, very few texts were adequately edited and the population in general was further cut off from its historical past. Since the Second World War, however, there has been an increased scholarly interest in Ottoman texts and in the transcription and edition of such texts. This interest has grown with the growth of a tolerance for some reemergences of older ethical, religious, and cultural practices and attitudes.

It is clear that the particular situation in Turkey today lends itself to the adoption of editing methodologies that take advantage of computer technology: there is a large cadre of well-educated persons with very positive attitudes toward technological innovation; the Latin alphabet is used [with modifications for Ottoman transcription]; most of the basic editing work remains to be done; there is already great interest in the types of concordancing, indexing, lexicographical analysis, etc. that can be most easily done by computers. Nonetheless, Ottoman studies is still an area that attracts persons who would be least likely to welcome technological innovation and so any change would need to bring immediate and obvious benefits. When the editing project was initiated, it was decided that the result should have the following characteristics:

1. It should be easy to use even for the most unsophisticated user.
2. It should be adaptable to many different circumstances and should be easily

supportable.
3. It should obviously eliminate the need for more than one entry of the basic text. [This is, of course, common to all computer word-processing systems but it is such a major departure from the usual round of draft typings that its benefits must be emphasized to those who have not experienced it.]
4. It should be capable of producing typeset camera-ready copy for printing. [This is a major potential benefit even in Turkey where the costs of more labor-intensive typesetting methods are growing rapidly.]

The project developed in several stages and was not without its problems and false starts. The first stage involved convincing a noted Turkish scholar and respected editor of Ottoman texts to come to the University of Washington to attempt to edit the collected poems of a 16th century Ottoman poet using the IBM XT already employed by the Department of Near Eastern Languages and Civilizations for the development of Turkish character-sets. Scholarly processes being what they are it turned out to be easier to bring the scholar than to have the necessary word-processing capabilities ready when he arrived. As a result, a rather cumbersome combination of Microsoft's WORD, Rosesoft's "smart key" program [PROKEY], and a series of BASIC programs developed by Robert Blum of the UW administration was used to enable the Turkish visitor to input and edit about 90% of the poems in the collection [over 500 poems] in about three months. The editor, who had had no previous experience of computers and no particular liking or aptitude for them, was an eager and willing convert to the process. Prior to his departure, we were also able to employ a simple translation program which converted the character-set designed for the XT to TEX notation and, subsequently, to produce a typeset sample of the edited text on the SUN minicomputer. The reaction of our visitor to the results of this process, which was carried out with the help of two fellow scholars without the intervention of typists or typesetters, was pure delight and amazement.

In the ensuing months the project has been considerably refined and improved. With the invaluable assistance of the UW Humanities and Arts Computing Center and its resident character-sets guru, Gerald Barnett, we have been able to develop a word-processing system that is simple, efficient, flexible, and low-cost. The system is based on Quicksoft's PC WRITE program used with EGA/VGA and compatable graphics hardware.* The advantages of PC WRITE for this kind of word-processing are numerous but it is worth mentioning a few in some detail.

Given the goal of making this technology widely available among scholars and students [especially among foreign students and scholars], the fact that PC WRITE is low-cost, share-ware [$89.00 with full support] makes it an attractive

* NOTE: At present, the system produces a host of irritating "ghost diacritics" when used with the IBM PS 2 graphics–these are a distraction more than a real hinderance but, as yet, we have no idea why they occur.

alternative. Moreover, PC WRITE permits virtually limitless customization of keyboards, fonts, printer controls, etc. in a manner accessible to persons without any knowledge of programming or programming languages. Using a simple set of programs–a program designed at Duke University for the creation of characters for display on an EGA driven monitor and a program being developed by Gerald Barnett of the UW for the production of downloadable printer fonts–we have been able to produce a word processing system that can display and edit an extended IBM character-set, which will allow the use of modern [roman alphabet] Turkish, the romanized transcription of Ottoman Turkish [Arabic alphabet], and a full English characters font. One can also switch instantly between a standard IBM keyboard, an IBM keyboard adapted to Turkish characters, and the standard Turkish keyboard with extensions for the Ottoman character-set. In addition, the system supports draft printing on the IBM Pro Printer and letter quality printing on the NEC and Toshiba 24 pin printers [with the use of a bi-directional tractor].

The extended IBM character-set uses 8 of the special European characters, 32 special Ottoman transcription characters [on ASCII codes 192–223], and 9 special modern Turkish characters [on ASCII codes 225–233], as well as the full English set. All of the modern Turkish characters appear as characters on the modified standard keyboard. The Ottoman Turkish characters [standard English characters with diacritics] are called up by two-key sequences. For example, a "d" with a dot under it is produced by striking "/" followed by "d"; all other special Ottoman characters are produced by the same sequence ["/"+"character"]. Keyboard arrangement and the particular character used to call up the special characters can be easily modified to suit the preferences of the user.

One fortunate aspect of the print control features of PC WRITE for this project is that the print control program can be set up to support two different fonts for each character. Therefore, TEX notation can be provided as an alternative for each character and translation from the usual word-processor font to TEX notation can be done automatically by simply printing in the TEX input character set to another file. Because PC WRITE produces "clean" ASCII files, the material is immediately ready for typesetting in whatever TEX environment is being used.

Accented character sets in TEX

In the few years since the official release of TEX, a number of attempts have been made to adapt the program to languages other than English. The best known successes have depended on adaptations of the program itself, partly because the standard release of TEX can support only one system of hyphenation at a time, which makes a truly bilingual document quite difficult to produce. These adaptations may be broadly classed as program-based extensions of the language. The extension which is most obviously necessary is the addition of a primitive which can control the switch between one predigested hyphenation pattern and

another. Michael Ferguson's bilingual CNRS-TEX, which was initially developed for an environment in the province of Quebec, where French and English are constantly intermingled, is one of the outstanding developments in this class of adaptations, and there are others as well.

A second extension is needed to get around the problem of hyphenation in languages which make use of diacriticals and accents. The basic form of TEX will reject any word containing an accent from the evaluation routine which normally looks for acceptable hyphenation breaks. In effect, any word with an accent is treated as if it were an unbreakable horizontal box, and is not evaluated for hyphenation at all. This can make line-breaking very difficult, and several users of TEX have found it necessary to introduce a loop into the program so that accents and diacriticals will be stripped out just before the entry to the hyphenation routine, and then returned to their remembered positions after the discretionary hyphen nodes have been inserted into the word.

The disadvantage of both these systems is that the adapted program is no longer TEX. It is often possible to add the extra features in such a way that the resultant program will produce `DVI` files that are indistinguishable from those generated by TEX, but the extra features are not generally available on all systems which run TEX, and the user is often excluded, therefore, from some of the most popular small system versions of TEX.

An alternative solution to the problem of accented languages, though not of bilingual hyphenation patterns, is a font-based, rather than a program-based approach. Font characters may be generated with the accents already applied, and mapped into unused or little-used areas of the normal Computer Modern font table. If these characters are then supplied with an appropriate TEX `\lccode` value, the hyphenation loop will recognize them as part of a sequence capable of being hyphenated. For a monolingual application in a language which makes intensive use of accents and diacriticals, this can be an attractive approach, especially when there are reasons for wishing to preserve the ability to make use of small system versions of TEX. This is the approach we have taken for Turkish TEX.

Turkish provides a delightfully vivid set of examples of accentuation and hyphenation. The Latin-letter character set which has been in use since the orthographic reform of 1928 is extended, even in Modern Turkish, by means of a considerable number of diacriticals and accents. A diligent search through the modern dictionary will produce several five- and six-letter words in which every character is accented, and an intensive search might come up with words as much as nine letters long with every character accented. In critical editions of Ottoman texts, the number of accents more than doubles. Modern Turkish knows only the accented and unaccented pair of letters 's' and 'ş', but Ottoman Turkish has 's', 'ş', 'ṣ' and 's̲', which represent four completely distinct characters in the Arabic alphabet. The letter 'h' shows almost as much variety, and so do several others. Our Ottoman Turkish font has twenty-seven accent and letter composites, in

addition to the basic twenty-six simple Latin letters. Moreover, all composites can exist in upper case forms as well as in lower case.

When a character set is as heavily accented as this, it is desirable to make sure that the accents are positioned over their letters as exactly as possible. The `\accent` primitive in TEX does a remarkably good job of positioning accents, but it depends on a very general algorithm, and tends to place accents exactly centered over or under the affected character, no matter what the appearance of that character may be. Donald Knuth recognized this limitation in the very earliest stages of the development of TEX, and has consistently recommended that frequently used combinations of character and accent be developed as composite single images in the font. The center of a character is not always the best visual position for an accent; top accents should often be slipped just a bit to the right, and bottom accents just a bit to the left of the mechanically defined centerline of the character. Height and depth of accents are similarly subject to aesthetic judgement. The`\accent` primitive of TEX works very well indeed for sparsely occurring accentuation, but not so well when accents occur in every second word.

The problem of hyphenation in Turkish is even more striking. Turkish is known as an "agglutinating" language, which means, in effect that each discrete logico-syntactic qualification of a basic word is expressed in a single syllable tacked onto all the other syllables in the word. At the same time, it is a language in which consonant clusters are virtually unknown. A Turkish word is made up of simple open and closed syllables, of the form `cv` or `cvc`, and in native words there is not even the distinction between long and short vowels. The result is a language in which word-length tends to be greater than it is even in English, and where, as a result, hyphenation is often necessary. The hyphenation rules are inherited from the syllabification of Arabic. A syllable is assumed always to consist of an initial consonant (even when that consonant is no longer written) and to terminate in a vowel or in the next unvowelled consonant. This pattern is followed so absolutely that it is permitted to break up native Turkish suffixes. The plural suffix *-ler-* will be hyphenated as *-le-rine* in an environment where the `-cv-cv-cv` pattern predominates.

A set of hyphenation patterns for Turkish will therefore be quite simple to produce, but it will have no effect on most Turkish words unless something is done about the problem of accents. A word such as *çektirilebilecek* ought to provide six discretionary hyphenation nodes: *çek-ti-ri-le-bi-le-cek*, but the `\accent` primitive applied to the first letter will guarantee that the standard version of TEX gives up any attempt to hyphenate it at all.* If the initial letter

* The word is a future participle, and describes something as being capable of being extracted at some time in the future–like a tooth. A morphological division of the word would produce a very different hyphenation pattern, *çek-tir-il-e-bil-ecek*, with only five nodes.

'ç' were a single character in a special font, and were provided with an `\lccode` value, the `\accent` primitive would no longer appear, and the word could be evaluated for hyphenation.

Since the majority of TEX users will never have to deal with `\lccodes` at all, a word of explanation is in order here. TEX is designed to take care of the problems of typesetting in a general manner, independent of the language of the text to be set. The program recognizes that while many languages have paired upper and lower case character sets, not all do, and the order of the basic text character set may not be that of the Latin alphabet. For this reason, specific upper and lower case pairings are not built into the program, but are supplied by macro definitions in `plain.tex`. Like all other definitions in plain.tex, they may be replaced, and it is quite possible to dispense with plain.tex altogether, and substitute another basic format file such as `sadece.tex`, `franc.tex`, `einfach.tex` or `sketo.tex`. (Knuth insists, for obvious reasons, that the one thing you may not call it is "plain.tex.") If additional characters such as the accented letters of Turkish are made part of the basic input coding table, then they are likely to exist in upper and lower case pairs. Each lower case code is given itself as a lower case `\lccode`, and the code of its upper case equivalent as its `\uccode`. These can be used to force conversion from one case to the other, but the `\lccode` serves an additional purpose. When TEX enters the program loop which searches for discretionary hyphen nodes in each word, it first unpicks all ligatures such as **ffi** and then evaluates the resultant list from the beginning, working on any given word only so long as every character it finds has a valid `\lccode`. Any node that is not a simple character with a valid `\lccode` causes the routine to terminate; the sequence so marked is supplied with no discretionary hyphen nodes at all, and therefore cannot be broken by the line-breaking algorithm. This is what prevents hyphenation in the case of the Turkish word given above.

Input Code Interpretation

The Turkish text-editing system described above is driven from a keyboard mapped to conform as closely as possible to the standard Turkish typewriter keyboard. This mapping is not used directly in the design of the Ottoman Turkish font and, in its present form, is isolated from the actual TEX input. After the raw input has been corrected, it is passed through a filter which converts the accented characters into character pairs (or, in a very few instances, into TEX command sequences). These pairings are based on a proposal made more than ten years ago at the Orientalist Congress held in Paris, in 1974. Owing to the extraordinary richness of the Ottoman Turkish character set, it has been necessary to extend the old proposal, but it still retains the original principles, which are closely associated with the coding scheme used by the Onomasticon Arabicum project. The Onomasticon Arabicum uses a post-positive dot and a post-positive hyphen to indicate diacriticals, which is acceptable in a data-base of names, but not in continuous prose text. To provide the indications for Ot-

toman Turkish diacriticals, we have taken over the exclamation point '!', the equals sign '=', and the colon ':'.

The exclamation point is used for all the "emphatic" letters of the Arabic alphabet (the alphabet in which Turkish was written until 1928). These are the letters *Ḍad* (usually pronounced as 'z' in Turkish, and hence paired with a non-Arabic letter known as *Ż̇ad*), *Ṣad*, *Ḥa'*, *Ṭa'* and *Ẓa'*. The equals sign is used for all the consonants which are represented in Latin-letter transcriptions by a letter with a bar under, such as d̲ (*dhal*), more commonly written in Turkish as 'z̲', and also for vowels with a macron or, following the Turkish convention, a 'hat' accent, and similar forms, chosen like the cupped 'ğ', because the equals sign is visually closer than the colon is. (Moreover, the colon is needed for a different variety of the letter 'g'.) The colon is a catch-all for everything else, but works out rather well visually, as it happens. The three post-positives are not accents, but regular characters, which use the TEX convention of ligatures to invoke accented characters from the font, just as the second 'f' in the normal TEX 'ff' ligature pair does. If a standard Latin-letter character does not have an associated ligature table in the font, a following colon will be unaffected. Thus, the letter 'o', when followed by a colon will produce 'ö', but the letter 'e' when followed by a colon will produce 'e:'. The equals sign is returned to its normal function in math mode, and the colon and exclamation point can be invoked by the command sequences `\:` and `\bang` when the simple character will not work.

This set of conventions produces an input file which can, if necessary, be edited on a ordinary terminal lacking the special Turkish character features, and which a Turkish speaker can become accustomed to without too much difficulty. When coupled with a well-designed macro file and a rewritten hyphenation table, it provides the possibility of naturalizing a TEX environment into Turkish without any large investment in special purpose hardware and rewritten versions of non-standard (non-)TEX.

The Font

Donald Knuth's Computer Modern fonts come with a wide range of accents, which cover most of the requirements for Turkish. The only obvious lack is the flat cup which is used under both upper and lower case 'h' as an aesthetic variant for the simple bar under the letter. All the existing accents in Computer Modern are designed for consistency with the stroke-weights and proportions of the underlying alphabetic characters, and it is therefore very desirable to retain the details of this design in any associated font of accented characters. The vertical and horizontal positions may be altered and, for other languages than Turkish, the angle of acute and grave accents over upper case letters, but the basic proportions of each accent or diacritical remain unchanged. This is achieved by taking over the entire text of the Computer Modern character file `accent.mf` and converting the `beginchar ... endchar` pairs to `def` and `enddef`. It is not quite so easy as that, but the process is essentially mechanical, and

guarantees the preservation of all the essential design details for each accent. (The flat cup under 'h' is based on the slavic tie accent, turned upside down.) The resultant file, `accdef.mf`, is now full of "definitions" which can be invoked as part of the program file for composite characters. Positioning, however, can not be entirely taken care of in the `accdef.mf` file. The accents in `accent.mf` are, for the most part, designed with a fixed reference point at the top of the image, but correct positioning usually requires a knowledge of where the bottom edge will be. It is herefore necessary to take some of the calculations from the accent definitions, and incorporate them into the description of the underlying character. For example, the superscript dot accent in tne Computer Modern font is produced as follows.

```
iff ligs > 0: cmchar "Dot accent";
numeric dot_diam#; dot_diam# = max(dot_size#, cap_curve#);
beginchar(oct "137", 5u#, min(asc_height#, 10/7x_height# + .5dot_diam#), 0);
define_whole_blacker_pixels(dot_diam);
italcorr h# * slant + .5dot_diam# - 2u#;
adjust_fit(0, 0);
pickup tiny.nib; pos1(dot_diam, 0); pos2(dot_diam, 90);
x1 = x2 = .5w; top y2r = h + 1;
if bot y2l < x_height + o + slab: y2l := min(y2r - eps, x_height + o + slab + .5tiny); fi
y1 = .5[y2l, y2r]; dot(1, 2);                                          % dot
penlabels(1, 2); endchar;
```

The corresponding `accdef.mf` definition is

```
def dot_accent(suffix $, @)(expr dotY_shift) =
save @;
forsuffixes $$ = @, @_: transform $$; endfor
numeric dh#; dh# := min(asc_height#, 10/7x_height# + .5dot_diam#);
define_whole_blacker_pixels(dh, dot_diam);
pickup tiny.nib; pos@_1(dot_diam, 0); pos@_2(dot_diam, 90);
x@_1 = x@_2 = x$; top y@_2r = dh + 1;
if bot y@_2l < x_height + o + slab: y@_2l := min(y@_2r - eps, x_height + o + slab +
.5tiny); fi
y@_1 = .5[y@_2l, y@_2r];
numeric dot_span; dot_span = dh - bot y@_2l;
@ = identity if dotY_shift <> 0: shifted(0, dotY_shift + dot_span) fi;
for n = 1, 2: forsuffixes e = l, , r:
  z@[n]e = z@_[n]e transformed @; endfor endfor
dot(@1, @2);                                                           % dot
penlabels(@1, @2); enddef;
```

To get this into position over the letter 'o', requires the following program text,

```
cmchar "The letter dotted o";
dot_sharp_values;
```

```
beginchar(oct "025", 9u#, dot_top#, 0);
italcorr 1/3[x_height#, asc_height#] * slant - .5u# if serifs: +.25dot_diam# fi;
adjust_fit(if monospace: .5u#, .5u# else: 0, 0 fi);
penpos1(vair, 90); penpos3(vair', -90);
penpos2(curve, 180); penpos4(curve, 0);
x2r = hround max(.5u, 1.25u - .5curve);
x4r = w - x2r; x1 = x3 = .5w; y1r = x_height + vround 1.5oo; y3r = -oo;
y2 = y4 = .5x_height - vair_corr; y2l := y4l := .52x_height;
penstroke pulled_arce(1, 2) & pulled_arce(2, 3)
   & pulled_arce(3, 4) & pulled_arce(4, 1) & cycle;                    % bowl
numeric dot_shift, dot_top;
define_whole_blacker_pixels(dot_diam, dot_top);
dot_shift = 0;                  % in this case, the position happens to be correct
x7 = x1 - .8dot_diam; x8 = x7 + 1.6dot_diam;
dot_accent(7, a, dot_shift);
dot_accent(8, b, dot_shift);
penlabels(1, 2, 3, 4, 7, 8); endchar;
```

in which the line

```
dot_sharp_values;
```

expands to a macro

```
def dot_sharp_values =
numeric dot_diam#; dot_diam# = max(dot_size#, cap_curve#);
numeric dot_top#; dot_top# = min(asc_height#, 10/7x_height# + .5dot_diam#);
enddef;
```

which repeats some of the calculations made in the definition of the dot accent.

The composites that result from this programming effort look, for the most part, identical to the results of the application of the **\accent** primitive to characters in the regular Computer Modern fonts. The one major difference comes in the shape of the "hat" accent over the letter '**i**'. In this instance, the accent would spread beyond the left and right side bearings of the underlying character and mess up the letter spacing if it were not pinched in, so a special narrow hat accent is provided for '**i**'. The proportions of each stroke remain essentially the same as those in the original model, but they form an acute angle over the top of the letter. Except in the case of this character and some of the uniquely Turkish dotted uppercase letters, it will probably be difficult to distinguish the two styles of accent in the final printed version even when they are intermingled in the same text.

The creation of the composite characters is only the first stage in the development of the font. Next, the italic correction must be set for all the italic and slant fonts. This is the spacing that may be added to the right side of any slanted character to prevent it from running into something like a non-slanted closing parenthesis. There does not seem to be any way except visual inspection

to discover an acceptable italic correction. One wants a fairly simple, general calculation, but one which will do rough justice to all slanted versions of the character. There were more proof copies generated to get the italic correction right than for any other feature of the font. (In the absence of any accessible system on which proofs could be displayed on the screen, a great many paper proofs had to be generated.)

Following this comes the generation of ligature and kerning tables, which are necessarily quite large, and need to be carefully worked out since there is only a finite region of a `tfm` file that can be devoted to them. The smaller of the two ligature tables, for the italic fonts, is shown in appendix A. It still needs one further refinement; the kernings appear in the order of English letter frequency, and it might be possible to gain a little efficiency by rearranging some of them. Notice that the '**f**' ligatures are altogether eliminated. In Turkish it is essential to retain the distinction between the dotted and the undotted '**i**, which cannot be done if the '**fi**' ligature is used. The problem that arises, in fact is to provide adequate separation between the dotted '**i**' and a preceding '**f**'.

In addition to the accented characters, it was necessary to design three additional characters for Ottoman Turkish. The simplest is a dot at about the bar height of lower case '**e**'. This is used for a type of Persian suffix known as "izafet," which is very common in Ottoman texts. The remaining two characters are representations of the Arabic letters "Ayn" and "Hamza," which are conventionally represented by opening and closing single quotes in most fonts. The "lazy man's *'ayn*" (as just illustrated) is acceptable for the occasional reference, but not for extensive literary texts. Ayn is not an accent, it is a regular consonant of the Arabic alphabet and Hamza, though it can be omitted in many positions is also a consonant. What is needed is a pair of characters which are clearly distinguishable from single quotes, but sufficiently like them to conform with the general appearance of Computer Modern. The programs shown below, draw on the same standard definition as is used to generate the single quotes, but alter the position and the proportions. The bulb is uppermost in both instances, and is somewhat smaller than the bulb of the close quote. The tail is brought out further from the side of the bulb, and is tucked more tightly under. The versions for slanted and italic fonts use some special transformations to insure that the *'ayn* (that was the character from the Ottoman font) is correctly formed. In effect, the character is built out to the left of the centerline, with a reverse slant, and then reflected back into the normal letter space. The program for these characters is Appendix B.

A complete passage from our first proposed critical text edition is given below, first in TEX input coding, and then as typeset. The text from which this passage was extracted runs to twelve pages, and was set without the benefit of a properly rewritten hyphenation table. By good luck, most of the English pattern hyphenations turned out to correspond with acceptable Turkish hyphenations, but it will certainly be necessary to make up a proper Turkish hyphenation table

in the near future. When that is done, and an appropriate set of formatting macros has been written to isolate Turkish text from non-Turkish text and math mode, we will have a Turkish language adaptation of TEX which can be exported onto any small TEX system, with no alteration of the program whatsoever. The full range of standard Computer Modern font styles will be available, and will blend in perfectly with the normal unaccented library of Computer Modern fonts. We will not have a truly bilingual version of TEX, but for a predominantly Turkish language environment we will be offering a cheaper and more accessible monolingual font-based adaptation.

```
\'A=s:ik!lik! zama=ninda \'is:k! va=sit!asi ve s:eyda=lik! \'a=lemi:nde
s:evk! vesi=lesi:, vus!lat eyya=minda mah!abbet muk!tez:a=si, fi:ra=k!
gu:nleri:nde h!urk!at i:k!ti:z:a=si, baha=r mevsi:mi:nde s!oh!bet
germi:yyeti:, mah!bu=blar mecli:si:nde s:ara=b keyfi:yyeti:, ca=na=neler
i:bra=mi ve \'a=s:ik!lar i:k!da=mi ve fuz:ala= mus!a=h!abeti: ve \'uk!ala=
i:lti:fa=ti, ehl..i: di:ller rag:beti: ve t!a=li:bler mi:nneti: i:le
di:du:gu: ebya=t ve es:\'a=r, ki: her bi:ri:nu:n= lat!i=f ma\'a=ni=si:
ca=m..i naz!ma s:ara=b..i rengi=n ve s:i=ri=n h=aya=la=ti bezm..i:
s!afa=da nuk!l..i: s:ekkeri=n olup mu\'a=s:i:ra=n..i mecli:s..i: z=evk!
bu meyh=a=nenu:n= ba=deci:si: ve h!ari=fa=n..i bezm..i: s:evk! bu
ka=s:a=nenu:n= sebu=-kes:i: olmis:lardi.  K!alem..i: i:\'ti:z=a=r bu
h=a=me..i: i:nki:sa=r i:le bu evra=k!a tah!ri=re i:k!da=m ve bu
ecza=ya tast!i=re i:hti:ma=m go:sterdi:.
```

ʿÂşıḳlıḳ zamânında ʿışḳ vâsıṭası ve şeydâlıḳ ʿâleminde şevḳ vesîlesi, vuṣlat eyyâmında maḥabbet muḳteżâsı, firâḳ günlerinde ḥurḳat iḳtiżâsı, bahâr mevsiminde ṣoḥbet germiyyeti, maḥbûblar meclisinde şarâb keyfiyyeti, cânâneler ibrâmı ve ʿâşıḳlar iḳdâmı ve fużalâ muṣâḥabeti ve ʿuḳalâ iltifâtı, ehl-i diller raġbeti ve ṭâlibler minneti ile didügü ebyât ve eşʿâr, ki her birinüñ laṭîf maʿânîsi câm-ı naẓma şarâb-ı rengîn ve şîrîn ḫayâlâtı bezm-i ṣafâda nuḳl-i şekkerîn olup muʿâşirân-ı meclis-i ẕevḳ bu meyḫânenüñ bâdecisi ve ḥarîfân-ı bezm-i şevḳ bu kâşânenüñ sebû-keşi olmışlardı. Ḳalem-i iʿtiẓâr bu ḫâme-i inkisâr ile bu evrâḳa taḥrîre iḳdâm ve bu eczâya tasṭîre ihtimâm gösterdi.

Appendix A
The `turkit.mf` driver file

```
                                    % Turkish Text Italic with full diacriticals, based on
                                % The Computer Modern Text Italic family (by D. E. Knuth, 1979–1985)
                                    % Adapted for Turkish by P. A. MacKay, January, 1987.

if ligs > 0: font_coding_scheme := "TeX Turkish"
else: font_coding_scheme := "TeX typewriter Turkish" fi;

mode_setup; font_setup;

italquery = oct "077";              % use the italic questionmark for this class of fonts
if not monospace: izafet_dot = oct "175"; fi

input tkital;                       % itall.mf lower case (minuscules) with undotted i
input romanu;                                   % upper case (majuscules)
input itald;                                    % numerals
input tkdotu;                       % upper case with dotted diacriticals
input tkdtil;                       % lower case italic with dotted diacriticals
input tkaccu;                       % upper case with assorted acents
input tkacil;                       % lower case italic assorted accents
input aynhmz;                       % ayn and hamza, izafet dot iff not monospace
input tkpnct;                       % punctuation common to roman and italic (reduced set)
if ligs > 0: input comlig;          % ligatures common with roman text
else: input tksub; fi               % substitutes for ligatures

                                % A thoroughly mixed list of names for the accented characters.
                            % These follow English, Arabic and Turkish conventions rather arbitrarily
                        % Only the letters that appear more than once in the ligtable are coded here.

HAT_A = oct "044"; CHIM = oct "013"; DAD = oct "000"; DHAL = oct "014";
CUP_G = oct "015"; GHAYN = oct "001"; QAF = oct "004";
DOT_O = oct "005"; TTA = oct "007";
DOT_U = oct "010"; HAT_U = oct "046";
hat_a = oct "074"; chim = oct "033"; dhal = oct "034"; dad = oct "020";
cup_g = oct "035"; ghayn = oct "021";
hha = oct "022"; kha = oct "036"; dot_i = oct "023"; hat_i = oct "075";
qaf = oct "024"; gnaf = oct "037"; dot_o = oct "025"; hat_o = oct "043";
tta = oct "027"; dot_u = oct "030"; hat_u = oct "076";

font_slant slant; font_x_height x_height#;

                                % Accent ligatures not complicated by questions of kerning
```

```
                                    % good for both monospace and variable-space fonts.
ligtable "C": ":" =: CHIM;
ligtable "G": ":" =: GHAYN, "=" =: CUP_G;
ligtable "H": "=" =: oct "016", "!" =: oct "002";
ligtable "I": ":" =: oct "003", "=" =: oct "045";
ligtable "N": "=" =: oct "017";
ligtable "S": ":" =: oct "052", "=" =: oct "0136", "!" =: oct "006";
ligtable "U": ":" =: DOT_U, "=" =: HAT_U;
ligtable "Z": ":" =: oct "011", "=" =: oct "0137", "!" =: oct "012";

ligtable "a": "=" =: hat_a;
ligtable "i": ":" =: dot_i, "=" =: hat_i;
ligtable "g": ":" =: ghayn, "=" =: cup_g;
ligtable "h": "=" =: kha, "!" =: hha;
ligtable "k": "!" =: qaf;
ligtable "t": "!" =: tta;
ligtable "s": ":" =: oct "053", "=" =: oct "0176", "!" =: oct "026";
ligtable "u": ":" =: dot_u, "=" =: hat_u;
ligtable "z": ":" =: oct "031", "=" =: oct "0177", "!" =: oct "032";

if monospace: font_normal_space 9u#;          % no stretching or shrinking
  font_quad 18u#;
  font_extra_space 9u#;
  letter_fit# := letter_fit := 0;
  ligtable "A": "=" =: HAT_A;
  ligtable "D": "=" =: DHAL, "!" =: DAD;
  ligtable "K": "!" =: QAF;
  ligtable "O": ":" =: DOT_O;
  ligtable "T": "!" =: TTA;

  ligtable "c": ":" =: chim;
  ligtable "d": "=" =: dhal, "!" =: dad;
  ligtable "n": "=" =: gnaf;
  ligtable "o": ":" =: dot_o;                 % no hat_o here
else: font_normal_space 6u# + 2letter_fit#;
  font_normal_stretch 3u#; font_normal_shrink 2u#;
  font_quad 18u# + 4letter_fit#;
  font_extra_space 2u#; fi

if not monospace:

  k# := -.5u#; kk# := -1 5u#; kkk# := -2u#;     % three degrees of kerning

                        % The following ligtable entries are based on the entries in
                      % roman.mf. It has been necessary to extract many parts of the
                        % original entries in order to keep the ligature structure clear.

  ligtable "d": "=" =: dhal, "!" =: dad,
    dhal: dad: "w": "l": "l" kern +u#;
```

```
ligtable "F": "V": "o" kern kk#, dot_o kern kk#, hat_o kern kk#,
    "e" kern kk#,
    "u" kern kk#, dot_u kern kk#, hat_u kern kk#,
    "r" kern kk#, "a" kern kk#, hat_a kern kk#,
    "A" kern kkk#, HAT_A kern kkk#,
  "K": "!" =: QAF,
  QAF: "X": "O" kern k#, DOT_O kern k#, "C" kern k#, CHIM kern k#,
    "G" kern k#, GHAYN kern k#, CUP_G kern k#, "Q" kern k#;

ligtable "T": "!" =: TTA, TTA: "y" kern kk#,
  "Y": "e" kern kk#, "o" kern kk#, dot_o kern kk#, hat_o kern kk#,
    "r" kern kk#, "a" kern kk#, hat_a kern kk#,
    "u" kern kk#, dot_u kern kk#, hat_u kern kk#,
  "P": "W": "A" kern kk#, HAT_A kern kk#;

ligtable "D": "=" =: DHAL, "!" =: DAD,
  DHAL: DAD: "X" kern k#, "W" kern k#, "A" kern k#, HAT_A kern k#,
    "V" kern k#, "Y" kern k#;
ligtable "O": ":" =: DOT_O,
  DOT_O: "X" kern k#, "W" kern k#, "A" kern k#, HAT_A kern k#,
    "V" kern k#, "Y" kern k#;

ligtable "A": "=" =: HAT_A,
  HAT_A: "R": "n" kern k#, gnaf kern k#,
    "l" kern k#, "r" kern k#, "u" kern k#, dot_u kern k#, hat_u kern k#,
    "m" kern k#, "t" kern k#, tta kern k#,
    "i" kern k#, dot_i kern k#, hat_i kern k#,
    "C" kern k#, CHIM kern k#, "O" kern k#, DOT_O kern k#,
    "G" kern k#, GHAYN kern k#, CUP_G kern k#,
    "h" kern k#, kha kern k#, hha kern k#, "b" kern k#,
    "U" kern k#, DOT_U kern k#, HAT_U kern k#,
    "k" kern k#, qaf kern k#, "v" kern k#, "w" kern k#, "Q" kern k#,
  "L": "T" kern kk#, TTA kern kk#,
    "Y" kern kk#, "V" kern kkk#, "W" kern kkk#,
  "b": "e": "p": "r": "e" kern -u#,
    "a" kern -u#, hat_a kern -u#,
    "o" kern -u#, dot_o kern -u#, hat_o kern -u#,
    "d" kern -u#, dhal kern -u#, dad kern -u#,
    "c" kern -u#, chim kern -u#,
    "g" kern -u#, ghayn kern -u#, cup_g kern -u#, "q" kern -u#;

ligtable "c": ":" =: chim,
  chim: "e" kern -u#,
    "a" kern -u#, hat_a kern -u#,
    "o" kern -u#, dot_o kern -u#, hat_o kern -u#,
    "d" kern -u#, dhal kern -u#, dad kern -u#,
    "c" kern -u#, chim kern -u#,
    "g" kern -u#, ghayn kern -u#, cup_g kern -u#, "q" kern -u#;
```

```
  ligtable "o": ":" =: dot_o, "=" =: hat_o,
    dot_o: hat_o: "e" kern -u#,
      "a" kern -u#, hat_a kern -u#,
      "o" kern -u#, dot_o kern -u#, hat_o kern -u#,
      "d" kern -u#, dhal kern -u#, dad kern -u#,
      "c" kern -u#, chim kern -u#,
      "g" kern -u#, ghayn kern -u#, cup_g kern -u#, "q" kern -u#;

  ligtable "n": "=" =: gnaf, "'" kern kkk#, gnaf: "'" kern kk#;

  ligtable oct "040": oct "100":                    % kerns for ayn and hamza
      "A" kern kkk#, HAT_A kern kkk#,
      "a" kern k#, hat_a kern k#;

  ligtable ".": "." =: oct "175";                                % izafet dot

  ligtable "f":                % Turkish cannot use the ordinary ligatures for "f".
      hat_i kern 3u#, dot_i kern 3u#,          % and therefore needs these kerns
      oct "040" kern 3u#, oct "100" kern 3u#;
fi
                                % ligatures for "-", "`", and "'" in the comlig file
bye
```

Appendix B

The `aynhmz.mf` file

```
% Ayn and Hamza (smooth and rough breathing) for Computer Modern
% These letters were originally coded by P. A. MacKay in December, 1986,

% Ayn and Hamza are based on the comma supplied in cmbase.mf,
% but the jut is increased, the tail is tucked in more tightly, and the
% bulb is uppermost in both cases. Ayn is NOT the mirror image of Hamza in
% slanted or italic fonts, thanks to a little magic with currenttransform.

% Character codes @040 and @100 are generated.

% if izafet_dot is known, character @175 (dot at bar_height) is generated
if unknown accsub: input accsub fi                    % needed for izafet_dot

newinternal pslant, nslant; pslant := slant; nslant := -slant;

cmchar "Rough breathing or `Ayn";
beginchar(oct "040", 5u#, min(asc_height#, 10/7x_height#), 0);      % height of i-dot
italcorr asc_height# * slant + .5dot_size# - 2u#;
adjust_fit(0, 0);
currenttransform := identity slanted nslant
   yscaled aspect_ratio scaled granularity;              % build italic with reverse slant
x1 - .3dot_size = hround(.5w - .3dot_size); y1 + .3dot_size = h;      % smaller bulb
if monospace: comma(1, a, .6dot_size, .35u, vround .85comma_depth); % large comma
else: comma(1, a, .6dot_size, 1.75u, .85comma_depth); fi      % comma; increased jut
currentpicture := currentpicture reflectedabout((.5[l, r], h), (.5[l, r], 0));
currenttransform := identity slanted pslant
   yscaled aspect_ratio scaled granularity;              % restore normal font slant
penlabels(1); endchar;


cmchar "Smooth breathing or Hamza";
beginchar(oct "100", 5u#, min(asc_height#, 10/7x_height#), 0);      % height of i-dot
italcorr asc_height# * slant + .5dot_size# - 2u#;
adjust_fit(0, 0);
x1 - .3dot_size = hround(.5w - .3dot_size); y1 + .3dot_size = h;      % smaller bulb
if monospace: comma(1, a, .6dot_size, .35u, vround .85comma_depth); % large comma
else: comma(1, a, .6dot_size, 1.75u, .85comma_depth); fi      % comma; increased jut
penlabels(1); endchar;

iff known izafet_dot: cmchar "Period raised to bar height";
dot_sharp_values;
beginchar(izafet_dot, 5u#, x_height#, 0);
```

adjust_fit(0, 0); **pickup** *fine.nib*;
define_whole_blacker_pixels(*dot_diam*);
pos_1(*dot_diam*, 0); pos_2(*dot_diam*, 90);
lft x_{1l} = hround(.5w − .5*dot_diam*);
y_1 + .5*dot_diam* = vround(*bar_height* + .5*dot_diam*);
$z_1 = z_2$; *dot*(1, 2); **penlabels**(1, 2); **endchar**;

What Should We Do for Japanese TEX? An Overview of Japanese TEX Systems

NOBUO SAITO, KAZUHIRO KITAGAWA

Department of Mathematics
Faculty of Science and Technology
Keio University
3-14-1, Hiyoshi,Kohoku
Yokohama,223 Japan
ns%keio.junet@japan.cs.net
kaz%keio.junet@japan.cs.net

1. Overview

There are several Japanese versions of TEXs available for the moment. For example, there are at least four versions of the Japanese TEX system: Canon, IBM, NIT, ASCII Corp. and Keio University. In general, there are three major adaptation methods to handle Japanese language: macro processing, preparing preprocessor, and extension of TEX itself.

1.1 NTT ECL

The first version is implemented with the macro processing method. This method is easy to develop, but there are a lot of limitations, and it is inefficient.

1.2 Canon and Japan IBM TRC (Tokyo Research Center)

They independently developed preprocessors that translate text written in Japanese into normal ASCII text with font identification. There is still a limitation for full utilization of the TEX system through the preprocessors. Advantages of these two methods are that Japanese versions of TEX can be developed without changing the TFM, or font format, and without a big change in device drivers.

1.3 ASCII Corp. and Keio University

We extended TEX itself as Prof. Knuth suggests in "TEX: The Program", and we exploited the SET2 dvi code that Knuth reserved for the future use. The Japanese version of TEX developed by ASCII Corp. is based on the original

TEX82 written in Pascal. The one developed by Keio University is written in the C language (originally developed by Pat Monardo). This method is suitable for developing an efficient system, although this work might be very hard to finish.

Incorporating the way suggested by Knuth, there are two big problems which occurred: in paricular, TFM and font formats are too poorly designed to handle Japanese. These problems also occur in handling large character set languages such as Chinese. To solve these problems, we extended the TFM and font formats, and we also modified the device driver. So, this Japanese version of TEX supports quite the same facilities as the English version does, though it is essentially impossible to pass the original trip test. As a result of this experience, an executable image of TEX was found to be very big. Hence, it is difficult to port this TEX system into small scale computers like PC's.

There are three Kanji code systems currently available in Japan: JIS code, Shift JIS code and EUC (Extended Unix Code). All of the Japanese versions of TEX depend on Japanese codes. One can process only JIS, another can handle only Shift JIS, and so on. From TEX's point of view, there is no problem which code we select to use.

Judging from these experiences, it is easy to adapt the TEX system to handle oriental languages that have large character sets, such as Chinese, Japanese and so on. Incorporating any method, we get the same output with little difference. This difference depends on fonts and some other parameters.

2. Problems in Developing Japanese TEX

The main work of the adaptation of TEX to Japanese is almost done, but it does not mean that our work is completely finished. There are still a lot of problems for us to solve. What should we do next? Solutions to the following problems are still in experimental stages.

- Which font is standard Japanese Kanji character set for TEX?
- How do we extend the TFM file for large character sets, including font representation form?
- How do we pack large font sets for efficient access?
- How do we do vertical and right to left typesetting in general?
- Is there any PDL which handles a 2 byte font code?
- How do we smoothly input Kanji and ASCII English without frequent mode changes?
- Is the Knuth and Plass line breaking algorithm suitable for oriental languages?

Currently, we are investigating all of these. The extension of TFM file and font packing problems are in the final stage.

2.1 Standard Font

Probably, it is cumbersome to prepare over 6000 character fonts manually written in the METAFONT program. One printing company, DNP (Dai Nippon Printing), is ready to provide a good font set. We are using the DNP font now. Its quality is fairly good and its cost is low!!

2.2 TFM and Font Format

The current TFM format cannot represent font information for more than 256 characters. From the Kanji point of view, the width and the height of most of Kanji characters are basically the same. Only for the special combination of special characters, the width between the two characters is shrunk in half size.

Based on this characteristic of Kanji, the original TFM is not suitable for the Kanji character set, extending only fontinfo entry being useless. So, we prepared one table indexing to the fontinfo field. One is the default field (most cases) and the others are special fields that point to the fontinfo entry. The Id field is added to distinguish between the normal and extended TFM file for Kanji characters.

Another problem is how to store fonts. Typically, the amount of space required for one Kanji font set is about 2M bytes. So, the device driver cannot read Kanji fonts at one time in the same way as with English fonts. To reduce font storage space, a packing method should be used. But, with this method, we cannot randomly access one character efficiently because the size of the packed font data is not the same for all characters. A PXL file can provide random access, but it needs more space.

Combining pk and index format is still large, because the index becomes too large.

So, we need more elaborated packing mechanism for fonts. Solutions for this problem are still in experimental stage.

2.3 Vertical Typesetting

How do we do vertical typesetting? Does TEX build the vlist before the hlist is built?

2.4 PDL Problems

PDLs (Page Description Languages) which are currently available cannot handle 2 byte codes. They say that Adobe produces a Japanese version of PostScript. We hope a lot of vendors will support multi byte PDLs.

2.5 Input Problem

We edit Japanese text using a Japanese version of Emacs, vi and so on. These

are good editing facility, but provide a poor mechanism to mix Japanese with English. Sometimes we use "word processor" running on an office system and personal computers. Word processor supports a good Japanese input mechanism, but poor editing facilities. There is no editor with comfortable and smooth Japanese input mechanism. We must build a standard editor like Emacs for Japanese text input and editing. We are building a prototype system running under the Andrew window system.

2.6 Line Breaking Algorithm

The Knuth and Plass line breaking algorithm is good for two phase line building. Japanese line breaking is 4 phase. If necessary, we need a new line breaking algorithm.

3. Towards Elaborated Japanese Document Processors

TEX is public domain software, and it should be widely used even in Japan. Japanized TEX systems will be popularized more and more in the near future. Currently, a lot of word processors running under ordinal personal computers are widely used in Japan, and in these systems Kanji input and editing mechanisms are elaborate. For the average users, these systems are quite useful, but for professional users these word processors seem to be quite incomplete. It is necessary to develop fully sophisticated Japanese document processors.

We would like to enhance our Japanese TEX system to replace ordinal Kanji word processors. For that purpose, it is necessary to design it with a better user interface, elaborate Kanji input mechanism and high quality Kanji output system.

The final goal of the user interface seems to be to realize the WYSIWYG scheme, and recent development of similar systems, like Interleaf, Publisher, Frame Maker and so on, are very helpful for us to set our final target. Kanji input mechanisms for ordinal Kanji word processors are recently well developed, and there are a lot of input schemes. The Romaji-Kanji automatic translation scheme is widely used, and it is possible to introduce these mechanisms into Japanese TEX systems.

The quality of Kanji output is entirely dependent on Kanji fonts, and it is possible to develop a METAFONT program through the use of vectorized Kanji font data.

日本語 TEX: JTEX

YASUKI SAITO 斉藤康己

NTT Software Laboratories
3-9-11 Midori-Cho Musashino-Shi Tokyo 180 Japan
〒180 東京都武蔵野市緑町 3-9-11 NTT ソフトウェア研究所
Phone: +81 (422) 59-3585
ARPA: yaski%ntt-20@sumex-aim.Stanford.EDU
CSNET: yaski@ntt-20.ntt.jp
JUNET: yaski@ntt-20.ntt.junet

ABSTRACT

JTEX is an upward compatible extension of TEX, which makes it possible to typeset Japanese as well as English. In this report, I will first describe JTEX from the user's point of view in detail, then explain the essence of implementation briefly. Several examples are included to help you appreciate the output quality of JTEX and this report itself is of course typeset by JTEX.

JTEX from the User's Point of View

Preparation of the Input File

In JTEX you can use everything in TEX without modification. What is different from TEX is that you can enter Japanese characters* wherever you want. In math

* How to enter Japanese into machines is one of the biggest problems in Japanese text processing. Nowadays the most popular method is Kana-Kanji Translation. You enter the reading of Japanese phrase in Kana or in Romanized form and type "translate-key". The machine looks up a dictionary of the correspondence between reading and writing and replaces the reading by its proper writing. When there are several candidates for writing, you must choose one of them.

mode, however, you must use `hbox` because data structures for math mode are not extended to accept Japanese characters. Japanese font selection is as easy as that of English fonts and independent from the selection of English fonts.

Note the following points when you enter Japanese characters:

- A tiny skip of '0pt plus small amount' (`\jintercharskip`) is inserted between consecutive Japanese characters, except kinsoku processing, to allow line breaking between normal Japanese characters. (Kinsoku processing will be explained later.)
- Between an English and a Japanese character, a small skip is inserted automatically to improve the output quality. This glue is not inserted between control sequences and Japanese nor between math mode and Japanese. JTEX refrains from inserting the skip when one of the characters is a Kinsoku character.
- Single carriage return between Japanese characters is ignored. But double or more carriage returns start a new paragraph as in TEX.
- You can use Japanese characters in control sequence names. Japanese characters are treated as if they are in the same category as letters (although in reality they belong to different categories). So do not start a Japanese sentence just after the control sequence without inserting a space.

Example Input and Output

In this section two example input files and the output generated from them are shown. Note the use of font loading and selecting commands in example 1, and a Japanese control sequence name in example 2.

Example input file 1

```
\overfullrule=0pt
\nokinsoku‘ヨ
\font\gothic=cmssbx10 scaled \magstep1
\font\normal=cmr12
\loadjfont\jnormal=dm12
\use jroma of dm12
\jnormal\normal
\baselineskip=15pt
\centerline{\seventeendg インタラクションを記述する言語の構想}
\bigskip
\centerline{\twelvedg 斉藤康己}
\medskip
\centerline{\twelvedg(日本電信電話株式会社　ソフトウェア研究所)}
\bigskip
\noindent
{\twelvedg\gothic 1. まえがき}
\smallskip
知能の発達や学習の Key concept としてインタラクションがある。また最近
```

```
はプログラミング環境の重要な要素としてインタラクションということが色々
と議論されるようになってきた。しかしながら「インタラクションとは何か？」
に関していまだ明確な解答は得られていないのが現状である。本稿ではインタ
ラクションの実体をとらえるためには何を表現したらよいかという問題を
エディタというコンテクストで考える。まずは身近な所でエディタの設計等に
役立つ定式化を考え，将来的にはインタラクションを介した学習などへの応用
も予定している。[1]
\medskip
\noindent
{\twelvedg\gothic 2. インタラクションの諸側面}
\smallskip
本稿の目的は具体的なインタラクションの現場から，インタラクションを構成
する要素を抽出し，インタラクション記述言語で何を表現すべきかを明らかに
することにある。本節で各々の要素を列記し，説明を加える。
\medskip
```

Output produced from example input file 1

インタラクションを記述する言語の構想

斉藤康己

(日本電信電話株式会社　ソフトウェア研究所)

1. まえがき

知能の発達や学習の Key concept としてインタラクションがある。また最近はプログラミング環境の重要な要素としてインタラクションということが色々と議論されるようになってきた。しかしながら「インタラクションとは何か？」に関していまだ明確な解答は得られていないのが現状である。本稿ではインタラクションの実体をとらえるためには何を表現したらよいかという問題をエディタというコンテクストで考える。まずは身近な所でエディタの設計等に役立つ定式化を考え，将来的にはインタラクションを介した学習などへの応用も予定している。[1]

2. インタラクションの諸側面

本稿の目的は具体的なインタラクションの現場から，インタラクションを構成する要素を抽出し，インタラクション記述言語で何を表現すべきかを明らかにすることにある。本節で各々の要素を列記し，説明を加える。

Exampe input file 2

```
\def\cdse{Cd\allowbreak S$_x$\allowbreak Se$_{1-x}$\,$}
\def\mum{$\mu$m} \def\taus{$\,\tau_s\,$}
\def\tauc{$\,\tau_c\,$} \def\taur{$\,\tau_r\,$}
\def\節#1{\noindent{\tendg\bf #1}\smallskip}
\節{4-1 スイッチング時間}
共振器の減衰時間 \taus とこれまでに報告されている \cdse ドープガラ
スの緩和時間 \taur から以下のとおりスイッチング時間
\taus を計算し、実験結果と比較した。
まず共振器の減衰時間を求める。Y-52 を用いた非線形ファブリ・ペロー共振器
のフィネス $\,F\,$ の測定値は 10 であった。ここでは簡単のため、本共振器を
反射率 $R_{\rm eff}$ が 72\% のミラーで構成された損失のない共振器 (フィ
ネスが同じ) と等価であるとみなす。このとき共振器の減衰時間 \tauc は
$$\tau_c = {n_0 l+(L-l)\over c_0(1-R_{\rm eff})}$$
で与えられる。
ここで $l,\>L,\>n_0,\>c_0$ はそれぞれサンプルの厚さ、共振器の間隔、
サンプルの屈折率、光速である。$n_0,\>l,\>L$ をそれぞれ 1.54,
300\mum, 600\mum とすると \tauc は約 10psec となる。
また \taur=16psec$^{(2)}$ とすると、スイッチング時間 \taus は
$$\eqalign{\tau_s = &\sqrt{\tau_c^2 + \tau_r^2}\cr
= & 18\>\rm psec\cr}$$
と求まり、実験結果と良く一致することが確かめられた。
```

Output generated from expampe input file 2

4-1 スイッチング時間

共振器の減衰時間 τ_s とこれまでに報告されている CdS_xSe_{1-x} ドープガラスの緩和時間 τ_r から以下のとおりスイッチング時間 τ_s を計算し、実験結果と比較した。

まず共振器の減衰時間を求める。Y-52 を用いた非線形ファブリ・ペロー共振器のフィネス F の測定値は 10 であった。ここでは簡単のため、本共振器を反射率 $R_{\rm eff}$ が 72% のミラーで構成された損失のない共振器 (フィネスが同じ) と等価であるとみなす。このとき共振器の減衰時間 τ_c は

$$\tau_c = \frac{n_0 l + (L-l)}{c_0(1-R_{\rm eff})}$$

で与えられる。ここで l, L, n_0, c_0 はそれぞれサンプルの厚さ、共振器の間隔、サンプルの屈折率、光速である。n_0, l, L をそれぞれ 1.54, 300μm, 600μm とすると τ_c は約 10psec となる。また τ_r =16psec[(2)] とすると、スイッチング時間 τ_s は

$$\begin{aligned}\tau_s &= \sqrt{\tau_c^2 + \tau_r^2} \\ &= 18\ \text{psec}\end{aligned}$$

と求まり、実験結果と良く一致することが確かめられた。

Japanese Fonts Available in JTEX

Currently available japanese fonts are divided into two groups:

JIS Fonts

JIS fonts are generated from JIS C–6234 (24 dot font for printers). This font set has four sizes shown below and is in the public domain. They are 24, 36, 48 and 72 dots big and roughly correspond to 7pt, 10pt, 14pt and 20pt TEX fonts on 300dpi printers respectively.

`\jissml`	アメリカ合衆国において日本の貿易黒字が
`\jisstd`	アメリカ合衆国において日本の貿易黒字が
`\jisbig`	アメリカ合衆国において日本の貿易黒字が
`\jislrg`	アメリカ合衆国において日本の

DNP Fonts

DNP fonts are distributed by Dai Nippon Printing Co., Ltd.. This font set has two groups: DM series and DG series. 'D' stands for Dai Nippon, 'M' for Mincho style and 'G' for Gothic style. (We call a font similar to san serif bold 'gothic' in Japanese.) There are various sizes for each font and the point size is appended at the end of each font name. The following table lists all the non-magnified DNP fonts available in JTEX:

DM5	DM6	DM7	DM8	DM9	DM10	DM12	DM17	DM20
DG5	DG6	DG7	DG8	DG9	DG10	DG12	DG17	DG20

There are also magnified fonts, but not all magnifications are supported. You must have different dot sizes for each magnification but you can use one dot fonts for several places (for example 17 point font can be used as 12 point at magstep2 or 10 point at magstep3). So currently available magnifications and magnifications which can be covered by available fonts are as follows:

Available magnifications

("(n,m)" means that you can use n-pt font at magstep-m as this size.)

magstep	5pt	6pt	7pt	8pt	9pt	10pt	12pt	17pt	20pt
0	avail.	avail.	avail.	avail.	avail.	avail.	avail.	avail.	avail.
0.5			avail.			avail.	avail.	avail.	
1	(6,0)	(7,0)	avail.			(12,0)	(10,2)	(10,4)	
2	(7,0)		(10,0)			avail.	(17,0)	(10,5)	
3			(12,0)			(17,0)	(10,4)	(10,6)	
4			(10,2)			avail.	(10,5)	(10,7)	
5			(17,0)			avail.	(10,6)		
6			(10,4)			avail.	(10,7)		
7			(10,4)			avail.			

You also have the same set of magnified fonts for Gothic style. To use these fonts, you must pay 95000yen to DNP per CPU and printer for a set containing both Mincho and Gothic in various magnifications described here. There are different sets for 300dpi, 240dpi and 480dpi printers. Some of them are shown below (these are the fonts preloaded in JTEX):

`\sevendm`	アメリカ合衆国において日本の貿易黒字が
`\tendm`	アメリカ合衆国において日本の貿易黒字が
`\tendg`	アメリカ合衆国において日本の貿易黒字が
`\twelvedm`	アメリカ合衆国において日本の貿易黒字が
`\twelvedg`	アメリカ合衆国において日本の貿易黒字が
`\seventeendm`	アメリカ合衆国において日本の貿易
`\seventeendg`	アメリカ合衆国において日本の貿易

New Control Sequences in JTEX

This section describes newly introduced control sequences in detail. Some of the materials are taken from on-line help file (`jtex-help.tex`).

Japanese Font Loading

To load non-scaled Japanese fonts, use the command `\loadjfont` instead of the ordinary `\font` comand. The syntax is the same so if you want to load DG12 and want to name it 'foo', do:

```
\loadjfont\foo=dg12
```

To load scaled Japanese fonts, you must use a slightly different command**. Again the syntax is very similar to that of `\font`:

```
\loadsfont\foo=dm10 scaled \magstep5
```

These loading commands are defined in `jplain.tex` and are expanded to the ordinary loading commands (`\font`) of 15 subfonts.

If you want to use japanese roman alphabets or numerals, you must declare this fact using the following commands BEFORE actually selecting a font:

```
\use jroma of dm10
```

Similarly if you want to use JIS Russian alphabets or Greek alphabets or Keisen–Youso (Line segments), you must declare this as follows:

```
\use jrussian of dm10
\use jgreek of dg10
\use jkeisen of dm12
```

** These two similar commands will be merged to a single command in a future distribution. The same is true for `\use` and `\uses`, `\usesecondlevel` and `\usesecondlevels`. I just learned how to write macros which accept optional arguments.

Also second level (Daini–suijun) JIS kanji are not loaded by the previous loading commands. If you use a specific second level kanji you should declare this BEFORE selecting a font as follows:

```
\usesecondlevel{繹} of dm10
\usesecondlevel{朧} of dg10
```

If you want to declare the use of various subfonts for foreign alphabets or the use of second level kanji for scaled fonts, at the moment you must use the following different commands with 's' at the end (see the footnote on previous page):

```
\uses jroma of dm10 scaled \magstep5
\usesecondlevels{卍} of dm10 scaled \magstep2
```

Font Selection

Font selection is easy, you just use the name you give to that font when you load that japanese font. For example, after the loading command and several declarations:

```
\loadsfont\bigj=dm10 scaled \magstep5
\uses jroma of dm10 scaled \magstep5
\usesecondlevels{繹} of dm10 scaled \magstep5
```

you can select this font by `\bigj`.

Seven Japanese fonts listed below are already preloaded in `jplain.tex`. So use these names to select these fonts.

```
\loadjfont\sevendm=dm7
\loadjfont\tendm=dm10              \loadjfont\tendg=dg10
\loadjfont\twelvedm=dm12           \loadjfont\twelvedg=dg12
\loadjfont\seventeendm=dm17        \loadjfont\seventeendg=dg17
```

Selection of English fonts and Japanese fonts are orthogonal. That means your choice of a Japanese font does not affect the current English font. Think of it as if there are both English and Japanese current fonts.

JTEX's capacity for the number of usable Japanese fonts in a job is limited to about 10. Considering the fact that `jplain.tex` already loads 7 Japanese fonts, you can only use several more Japanese fonts in one job. With JLATEX the situation is much worse because `lplain.tex` preloads more English fonts.

Easy-to-do Style Selectors

Here style means the selection of various fonts and line spacing in essence. If what you want is to get the output in an appropriate size, use the following commands:

`\smalljapanesefont`:	gives you a 7pt output
`\standardjapanesefont`:	gives you a 10pt output
`\bigjapanesefont`:	gives you a 12pt output
`\largejapanesefont`:	gives you a 17pt output

How to Control Spacing

Spacing of Japanese characters is controlled by the value of the following glues:

`\jintercharskip`:
A glue inserted between Japanese characters except after post-kinsoku character or before pre-kinsoku character. This glue guarantees the breaking of japanese sentences at any point except kinsoku processing explained below.

`\jasciikanjiskip`:
A glue inserted between ASCII and Japanese characters, except for kinsoku characters. The value of this glue is normally set to a quarter of full Japanese space (`\jspaceskip`, see below) to make English words in a Japanese sentence and Japanese words in an English sentence look nice.

`\jspaceskip`:
A glue which is used when you specify JIS space in your source file. This is exactly the width of all Japanese characters in a font. You can use this to specify one character indentation at the beginning of ordinary Japanese paragraphs as follows:

```
\parindent=\jspaceskip
```

These three skips are assigned their values when you select a certain font. So if you want to change them, do so AFTER the font selecting command.

Kinsoku Processing

Kinsoku processing is a way to avoid the appearance of certain characters at the beginning of a line or at the end of a line. In JTEX this is accomplished by omitting the normally inserted `\jintercharskip` before the pre-kinsoku character and after the post-kinsoku character. A pre-kinsoku character cannot appear at the beginning of a line and post-kinsoku character cannot end the line. Pre-kinsoku and post-kinsoku characters are declared by `\prekinsoku` and `\postkinsoku` control sequences in `jplain.tex` and you can change them if you don't like the normal setting. Examples are:

```
\prekinsoku‘’
\postkinsoku‘‘
\prekinsoku‘っ
```

You can reset them by the use of the `\nokinsoku` command. You can only set kinsoku codes for ASCII characters, Japanese symbols (in subfont `\jsy`), Hiragana (`\jhira`) and Katakana (`\jkata`).

Terminal and File Type Setting

There are two control sequences (`\kanjiterminaltype` and `\kanjifiletype`) to specify the terminal and file type respectively. If you set the value of these switches to one of the following, JTEX can output the error messages and its output to files using the specified escape sequences.

0 ASCII only
1 Escape sequences are <esc>$@ and <esc>(J (Default)
2 Escape sequences are <esc>$@ and <esc>(H
3 Escape sequences are <esc>$@ and <esc>(B
4 Escape sequences are <esc>$B and <esc>(J
5 Escape sequences are <esc>$B and <esc>(H
6 Escape sequences are <esc>$B and <esc>(B
10 Shift JIS code (NOT YET IMPLEMENTED)
20 Extended Unix Code (NOT YET IMPLEMENTED)

JTEX from the Implementor's Point of View

Implementation Overview

I tried three different ways to implement Japanese TEX. First, so called Preprocessing method. Japanese characters in source file are converted to subfont selector and `\char` pair producing an intermediate file. This intermediate file is then given to ordinary TEX. I abandoned this method quickly because the existence of an intermediate file makes debugging difficult and processing speed was not so good. Secondly I implemented the same conversion from Japanese characters to (subfont, char) pair using only the macro facility of TEX. TEX's macro is powerful enough to allow this. But this macro version[1] is very slow, so the current JTEX is implemented by modifying TEX itself. (See [2] for more detail.) I tried hard to make modifications as minimum as possible so as to insure upward compatibility. About 1800 lines were added to the change file. Currently modified parts of "TEX: The Program" are as follows:

Part 1, 5, 12:	Basic utilities
Part 15, 17, 18:	Basic symbolic names and data structures
Part 20, 24, 25, 26, 27:	Token lists and basic scanning routines
Part 30:	TFM file handling
Part 46, 47, 49:	Chief executive (*main_control* routine)

I didn't change any data structures except that token list members are extended to include Japanese characters. Japanese characters are represented as (subfont, char) pair internally. JTEX swallows Japanese characters and converts them to (subfont, char) pair inserting necessary skips between each Japanese character and between ASCII and Japanse characters. After this input processing, everything is exactly the same as in TEX. JTEX notices the fact that it is processing Japanese only when it outputs messages or texts containing Japanese to terminals and to files. Otherwise JTEX breaks lines and builds pages using the same algorithm in TEX.

Subfonts

In Japan, JIS (Japanese Industry Standard) C–6226 code "Code of the Japanese Graphic Character Set for Information Interchange" is widely used to represent

Japanese characters in the computer. Characters in this code table are divided into 33 subfonts in JTEX. This division naturally corresponds to the categories in the code ("ku" in C–6226 table). The control sequence name for each subfont is used to refer to the individual characters in each subfont. Usually a user is not aware of the existence of subfonts, but if he wish, he can specify, say the second character in 4-ku, by "{\jhira\char2}".

\jsy	1-ku & 2-ku (symbols)
\jroma	3-ku (numerals & roman alphabets)
\jhira	4-ku (hiragana, phonetic symbols)
\jkata	5-ku (katakana, phonetic symbols for foreign words)
\jgreek	6-ku (greek alphabets)
\jrussian	7-ku (russian alphabets)
\jkeisen	8-ku (line segments)
\ja,...,\jl	16-ku,...,47-ku (2965 first level kanji)
\jm,...,\jz	48-ku,...,84-ku (3388 second level kanji)

Each subfont contains at most 256 characters, so GF and TFM file formats conform to TEX's font files.

Modification to TEX's Input Mouth

There are several ways to represent a file with both ASCII and JIS characters in it. JTEX accepts a file in which ASCII characters are represented as is and JIS codes (two 7bits bytes) are surrounded by escape sequences ("<esc>$@" or "<esc>$B" to start JIS code sequence and "<esc>(J" or "<esc>(B" to end it.) JTEX's input mouth converts two byte JIS codes to internal subfont number and character number, then sends them to JTEX's (TEX's) gut level. It is necessary to do this quickly, so the inner loop of TEX in *main_control* which munches ordinary characters was modified to do this.

Spacing and Kinsoku Processing

Ordinary Japanese sentences can break at any point between two consecutive characters. But there are exceptions such as before the punctuation marks or after the open parentheses. These exception rules are called "Kinsoku(禁則)" in Japanese. Kinsoku Processing is realized in JTEX as the omission of \jinter-charskip normally put automatically between every pair of Japanese characters. This processing is also effective between ASCII and Japanese character, and you can specify certain ASCII characters such as period or comma as Kinsoku characters.

Extra spaces after JIS period or comma must be handled using the equivalent of sfcode, but it is not yet implemented. In current Japanese fonts, every character including JIS symbols such as period, comma and various parentheses has the same width (\jspaceskip). JIS period, comma and close parentheses etc. are placed on the left hand side of a bounding box, so extra space is already there in the design and is not inserted using the similar trick as sfcode.

DVI File Produced by JTEX

The output of JTEX is a DVI file which is perfectly compatible with ordinary output of TEX. It does not use the "*set2*" DVI command provided for oriental languages. This means that you can give the output of JTEX to ordinary device drivers without modification. This is not always true in practice, so you need to tune your driver a little bit.

Modification to Device Drivers

We are using Imagen printers (8/300 and 3320) from a DEC2065 and several SUN3s. The driver on the DEC2065 is called DVIIMP written by Arthur Samuel. This driver works fine with small Japanese documents of a few pages but degrades swiftly with larger jobs. The main problem is the loading of the entire font information at the time when one character in that font is first encountered. Also the assignment of 'family' and 'member' to each character is rather inefficient particularly because there is no "working set" property in JTEX's subfonts ("Use of one character in a font implies the use of other characters in the same font for a while.")

On SUN3, we use a driver for imagen printers distributed with the Unix TEX distribution. Slight modification is also necessary because this driver uses PXL file and normal PXL files can contain only 128 characters. The PXL file format was extended to include up to 256 characters per font and the driver modified to accept extended PXL files.

Font Generation

All the Japanese fonts available in JTEX are dot based fonts. Public domain JIS fonts are generated from JIS 24 dot fonts by a simple LISP program. The same LISP program was used to generate GF and TFM files for DNP fonts from the various sized dot fonts provided by DNP. GF files for DNP fonts occupy about 91Mbytes and corresponding PXL files amount to 142Mbytes***. It is obvious that you need an efficient way to reduce the font file size.

JTEX Availability

JTEX is a public domain software and you can get it free with JIS fonts. In Japan, the Japan Society for Software Science and Technology distributes it. For users abroad, the easiest way is to get it from `Turing.Stanford.EDU` by anonymous FTP. The TOPS-20 version of JTEX is installed on this machine under `PS:<JTEX>`, so if you are on ARPAnet get JTEX from there. The UNIX version is also available from the author. I am thinking to make a public copy on some machine on ARPAnet, but until then ask me directly.

*** GF files for DM series fonts are about 48Mbytes and DG series, 43Mbytes. PXL files for DM and DG series fonts are 71Mbytes each.

To use DNP fonts, you need licensing with DNP. Send request to DNP directly or ask me. The person to contact at DNP is:

Tadashi Saito
3rd Section, Second System Development Dept.
CTS Division, Dai Nippon Printing Co., Ltd.
1–1–1 Kaga–cho Ichigaya Shinjuku-ku Tokyo 162 Japan

When you sign the agreement and pay 95000yen per set to DNP, you will get the font.

Future Work

There remains many things to be done if you consider JTEX as a total typesetting system.

- Some Japanese are written up to down. And we need to support it. But this is rather simple. Just rotate the font 90 degrees counterclockwise and adjust the centerline of each character if necessary.
- We need to build a collection of macro packages to facilitate the use of JTEX in various applications. Locally various forms are converted to JTEX format, and many JLATEX style files are written. It is necessary to organize these macros into a useful package.
- It may be necessary to enlarge the number of Japanese fonts usable in one document.
- Enhancement of Japanese fonts is really needed. To define all Japanese characters in METAFONT is a great challenge. And in a long run, someone or a group of people preferably consisting of both font designers and computer scientists must do it.

Conclusion

TEX is general enough to be extended to Japanese Typesetting as this work clearly shows. And extending TEX to Japanese is NOT difficult but NON–trivial. JTEX accomplished this NON–trivial task.

References

[1] Yasuki Saito: "Japanese TEX" (in Japanese), Working Group on Japanese Document Processing 10–3, IPJSS, (January 1987).
斉藤康己: "日本語 TEX", 情報処理学会日本語文書処理研究会予稿 10–3, (1987年1月).

[2] Yasuki Saito: "Report on JTEX: A Japanese TEX", TUGboat, vol.8, no.2, (July 1987).

Developing TEX DVI Driver Standards

ROBERT W. MCGAFFEY

Oak Ridge National Laboratory
Building 4500–S
P O Box X
Oak Ridge TN 37831–6144

When I encountered TEX at the Oak Ridge National Laboratory (ORNL) several years ago, I determined that the quality of output was such that it was the first mathematics that I had seen generated by a computer that I could not *tell* was generated by a computer. Next, I decided that I wanted to use TEX for my reports no matter what effort was required to use it. (This was fortunate because we had the old TEX78 which I am sure all of you know is extremely inferior to the present day TEX.)

At the time, laser printers were as scarce at ORNL as the `\iotabar` (which is now ι). So, I busied myself with the task of writing a driver for the 200-dpi Versatec. But then along came the new improved TEX and I had to start over. Then I persuaded my department head's two secretaries to try TEX. They tested it on one document, then dropped their previous typesetter (which shall remain nameless) and never looked back.

The next event in my story is the formation of a TEX support group here at ORNL. After much politicking such a group was set up and I was fortunate enough to be the project leader. I then decided that to write my own drivers for the many devices would be a mistake because the number of output devices was many and the time needed to write even one driver is long. We needed success right away.

To show you what we were up against: Here at the Laboratory, we have a CRAY, several IBM mainframes, some Data Generals, many VAXs, IBM PCs of many flavors, Macintoshs, and probably some others I'm not aware of. We do not expect to run TEX on all of these machines but certainly on most of them. For output devices we have an Apsμ5, Apple Laser Writers, HP LaserJets of both series, DEC LN03s, a QMS, Versatecs, and others I am sure. So I knew that we couldn't develop drivers in house.

Thus, I turned to the vendors for help, and they were glad to send us lots of information on their drivers and everything they could do for us. So I sat down with the information in an attempt to decide which drivers would handle our needs. And I discovered an interesting fact: vendors do not tell you what features their drivers do *not* have! Many of the drivers I read about do not mention that

they do not support landscape mode (and we need landscape mode). I got so involved in looking at all of the positive features of the drivers that I could no longer see their shortcomings.

A solution occurred to me. Why not write a description of the driver that we needed and then compare each available one to it. Then, not only would I know the missing features but would be able to talk intelligently to vendors on the phone. One thing lead to another and I decided it would be even nicer if the TEX Users Group (TUG) itself could put a little gentle pressure on vendors in our behalf. So rather than describe our (ORNL) ideal driver, I decided to describe *the* ideal driver.

Modesty forbids my saying that I did an excellent job. In fact, once I started writing the paper, I realized that I was not qualified to dictate what should be done, so I wrote the article (which appeared in TUGboat Volume 8, Number 2) with the idea in mind that it would generate discussion. It did. Barbara Beeton called me before the envelope was dry, and she was so enthusiastic that I found myself volunteering to head a committee to generate some driver standards. And here I am.

[*At this point in the talk I asked for questions and comments and was pleasantly surprised to discover that various TUG members had much to say on the subject. Some were very opinionated and said so in no uncertain terms. I would like to thank everyone who participated in the discussion which took place and give a special thanks to those who served on the panel for the talk and to Bart Childs for the support he gave to all of us.*]

I am looking forward to serving TUG in this capacity.

A TEX DVI Driver Family

NELSON H. F. BEEBE

Adj. Asst. Research Professor of Mathematics
Center for Scientific Computing
South Physics Building
University of Utah
Salt Lake City, UT 84112
USA
Tel: (801) 581-5254
ARPANet: BEEBE@SCIENCE.UTAH.EDU

ABSTRACT

A *portable family* of TEX DVI driver programs is described. These have been implemented under five different operating systems and eight different compilers for about twenty different output devices.

The goal is to support a large number of operating environments and output devices from a *single* set of source files.

Introduction

The TEX typesetting system now runs on several different personal computers, most major commercial minis and mainframes, and even on one supercomputer. TEX itself is written in Web, which is processed by two auxiliary programs, Tangle, to produce a Pascal program, and Weave, to produce a TEX input file which documents the software. If you have never examined a Web program, rush to your nearest bookstore or library for copies of Volumes B and D of Donald Knuth's book series *Computers and Typesetting* [KNUT86], where the programs

for TEX and METAFONT are beautifully presented.

TEX 's primary output is a compactly-encoded binary file of 8-bit bytes, called the DVI (DeVice Independent) file. It is the job of another program, called a DVI driver, to interpret this binary file and transform it into a device-*dependent* file which can be displayed on some particular output device, such as a dot-matrix printer, laser printer, bitmapped screen display, or high-resolution phototypesetter. In addition to reading the DVI file, this usually entails access to font files which contain encoded descriptions of character bitmaps.

Although TEX code is written in a subset of Pascal, there are nevertheless a few areas, such as the notorious missing **otherwise** clause in a **case** statement, and the opening, closing, and naming of files, which exhibit some operating system and/or compiler dependence. In order to avoid the need for changing the master Web source files, Tangle and Weave support the concept of a *change file*. This file contains line sequences that looks like

```
@x
⋮
original source lines
⋮
@y
⋮
replacement lines
⋮
@z
```

where the @x ... @y section contains enough original text to uniquely identify a fragment of the Web file, and the @y ... @z section contains the replacement for that fragment. The @x ... @y ... @z sections in the change file must match the order of the original text in the Web file. Since normally only one implementation of TEX is carried out on a particular operating system, and the resulting program then shared, or possibly sold commercially, few people ever have to write a Web change file.

For a DVI driver program, the situation is more complex. First, one driver is needed for each output device. Second, implementations of that driver will be needed for perhaps several different operating systems. Third, the program must be able to decode the complex font file formats, at least three of which are in common use. Fourth, efficient processing of the DVI file requires the ability to randomly seek to any arbitrary byte in the DVI file, and the font files, and begin processing from precisely that byte.

To give some flavor of the file sizes involved, the 494-page TEXbook has a TEX manuscript of 1.37Mb (megabytes), and the DVI file produced by TEX from it is about 1.96Mb, a 43% expansion. This is an average of 2.8Kb (kilobytes)

per manuscript page, and 4Kb per DVI page. Font file sizes vary significantly, depending on the encoding scheme, the character size, and the device resolution. For the popular 300-dot/inch laser printers and the compact PK-format encoding, 10pt font files are 4Kb to 7Kb in size. Most documents will require a dozen or two different fonts; the TEXbook requires 54!

Whereas N different operating systems only require N different implementations of TEX, getting visible TEX output for M different output devices will require $M \times N$ DVI drivers. It is in fact much worse than this, because there now exist many independently-developed DVI drivers, so we need to talk about a number more like $K \times M \times N$ drivers.

It is the thesis of this work that such an explosion is *completely unnecessary*, and with suitable care in the software development, it is possible to have *one* set of code files which simultaneously support several different operating systems, and many different output devices. A general enhancement added to this family then is immediately available for all supported operating systems, and all supported devices, merely for the price of recompiling and linking the code.

It is also the thesis of this work that because development of a DVI driver entails a substantial effort, there should be no need to repeat this effort at TEX sites all over the world. To that end, I have placed my work completely in the public domain. I have not even gone so far as to copyright the code, or to follow the GNU Project Manifesto in inserting a legal notice that you are welcome to obtain the software, use it, and give it to others, but only if the legal notice is retained in the software, and only if you distribute it further exactly as you obtained it, holding nothing back, and charging nothing for it. I do hope you read the GNU Manifesto (it is present in every copy of GNU Emacs, which is spreading rapidly over the world), and I do hope you will not charge people for work contributed by others. I also hope that if you find bugs, or make improvements, or have suggestions, you will communicate them to me so that everyone may benefit from our collective efforts. I cannot possibly enforce this wish, so it is up to you to help.

Remember that Don Knuth and Leslie Lamport have given us TEX, METAFONT, and LaTEX, and offered us the chance to change, for the better, I believe, the way in which we communicate in print.

The remainder of this article will describe some of the features of the DVI driver family, the programming language choice, the host environments in which it is known to run, and the devices it can produce output for. It will then discuss more detailed issues of code size, coding standards and portability, some case histories of problems that have been encountered, and the question of commercial versus public-domain software. It concludes with information about obtaining copies of the software, and gives credit to others who have contributed to the development.

Features

With burgeoning use of microcomputers, larger numbers of computer users will routinely work on multiple operating systems. I personally have daily contact with four or five operating systems, and three are generally available in different windows of my workstation.

To a very high degree, these DVI drivers all present the same user interface for all devices and for all operating systems. This is valuable for users, because no relearning is necessary. For the same reason, I insist on working with Emacs, or an Emacs-like editor, for text editing. For a touch typist, it simply is not worth the effort of learning another set of keystrokes, and anyway, I have never yet encountered an editor which comes remotely close to Emacs in power and convenience.

These drivers have several features that are worth listing here, because they are absent from many other drivers that have been developed elsewhere:

- uniform Unix-like command line interface on all machines for all output devices;
- control of page selection (-o<start>:<stop>:<incr>), and printing order (forwards or backwards) (-b);
- specification of the number of output copies;
- default and user-specified font substitution;
- support for GF, PK, and PXL font file formats;
- default directory paths overridden by environment variables and command-line options;
- user control of magnification;
- multiple DVI files can be processed in a single invocation of a driver.

It is critically important for a DVI driver to offer user control of output page selection, and since it has to contain a loop over document pages anyway, it is relatively trivial to add support for skipping pages which are not in a user-specified page range. Any number of page ranges can be specified, and an increment between the starting and ending pages can optionally be specified.

For example, consider printing a two-sided document on a printer which does not support duplex printing. Laser printers based on the popular Canon LBP-CX engine stack output face up, with the last page printed on top. Thus, a driver for such a printer would by default process pages in reverse order. One invocation with the option -o1:9999:2 will produce an output file with the odd-numbered pages, which when printed would have page 1 on top. The printed pages can then be reinserted in the input tray face up, page 1 on the top, exactly as they were found in the output tray. A second run with -b -o2:9999:2 would then print even-numbered pages on the backs of corresponding odd-numbered pages; the -b option requests backwards order on the second run. The output document will appear in the tray with page 1 face down on the bottom, correctly printed on both sides.

Laser printers function by creating, in internal memory, a page image which is transferred to a photo-sensitive drum that passes over a toner reservoir; the toner on the drum image is then heat fused onto the paper surface. At that point, all of the busy work has been done, and the engine can continue to produce copies of the image at its rated printing speed. Since the drivers permit a copies count option, it takes no more host CPU time to get multiple copies of a document. Of course, the lower-cost engines do not provide multiple output bins, so one still has to sort the output by hand. In an office environment, a -c2 option could routinely be used to get file copies of printed correspondence.

When TEX runs, it has font character sizes either preloaded, or loaded dynamically from TEX Font Metric (.tfm) files. Presumably, the corresponding bitmap font file will be present on the system as well, and the DVI driver will have available all of the necessary fonts for printing. However, in a distributed computing environment, this may not be the case, and it is absurd that some drivers abort processing when they cannot find a referenced font. My driver family attempts to deal with this gracefully.

If the user takes no other action, the driver will consult a built-in magnification table to attempt to find a font in the same family at a nearby magnification, and use that. A warning message is issued when that happens, but the user at least will get output which approximates what was expected. If no such neighbor can be found, and no font substitutions have been requested, processing continues with assumed zero-size characters for that font. The drivers do not yet attempt to find and read the .tfm file, and produce, say, a rectangular outline or shaded area of the correct size; that work remains on my to-do list.

It is possible, however, for the user to specify explicit font substitutions, either for a specific magnification, or for an entire font. An example of the latter would be to substitute cmr10 when the document actually referenced amr10, perhaps because it used an old macro package which has not been brought up-to-date with the naming conventions of new METAFONT. Such substitutions are provided in a file that can be specified on the command line, or if this is not done, a file with the same name as the DVI file, but extension .sub, will be tried. If that fails, then a file texfonts.sub will be tried in the current file directory, and if that too fails, then a file of the same name in the standard TEX input directories will be tried. This provides for document-specific, user-specific, and system-wide font substitution fallbacks.

The directory paths that are searched have default values which are set at compile time, but these can be overridden at run time by command-line options, or global environment variables or logical names.

Font magnification is available as a command-line option; it can be specified either as a normal TEX magstep value, such as -m0.5 to print at magstep half (about 9.5% larger), or as a large integer value based on the old convention of 1000 meaning 200 dots/inch. On a 300-dot/inch laser printer, -m1643 would also be magstep half. Negative magsteps are permissible as well, to produce reduced

output. User control of magnification is convenient when documents must be prepared for reproduction at a different size, as is common in the publishing industry, and is also useful for visually-impaired readers who may need larger type. Of course, such magnifications assume the availability of fonts at those sizes, but that is the responsibility of the local installation to provide them. The drivers do not make any attempt to resample font bitmaps to rescale them, such as xdvi in the X-windows system does. With bi-level fonts (i.e., dots are either on or off), which is all that METAFONT produces, such rescaling from relatively low-resolution masters is likely to produce character bitmaps of marginal quality. Still, if someone contributes code to do this, it could be incorporated as an optional feature.

Why C and Not Web?

This driver family is written in the C programming language [KERN78], not Web. Since Web is the language used for TEX, METAFONT, Tangle, Weave, and other pieces of TEXware from the TEX project, and for what Don Knuth has called "literate programming" [KNUT84], I believe that some justification is necessary for the language change.

The output of Tangle is a Pascal program which must be compiled and linked. Pascal was developed by Niklaus Wirth in 1968, and the first compiler for it was operational in 1970. The language was based on Algol 60 and Algol W, but introduced new features to better support Wirth's paradigm that *Algorithms + Data Structures = Programs*, which he used for the title of his 1976 book [WIRT76]. Until the ANSI/ISO Standard was adopted [ANSI83b], the language was defined by Jensen and Wirth [JENS74]. It is a small strongly-typed language that permits rapid one-pass compilation, but is nevertheless rich enough to permit its use for teaching healthy programming habits and thought patterns. It has been eminently successful for that purpose, but it is lacking in several areas.

> Virtually *every* implementation of Pascal has to extend it in some way, since standard Pascal (as described in Jensen & Wirth) is absolutely unusable, and ISO Pascal is not much better, ... resulting in a tower of Babel of dialects that is surpassed only by the BASIC language.
>
> *E. Wayne Sewell, TUGBoat, 8, 119 (1987)*

Pascal has a peculiar I/O model that does not adhere to the traditional *open—process—close* sequence; instead, it has a **reset(file)** statement that opens a file for input and then reads the first item, and a **rewrite(file)** statement that opens a file for output, and truncates it to zero length. This of course creates havoc with interactive programs that output a prompt, then read some input, since the standard input and output files have already been opened by implicit **reset** and **rewrite** statements, and the program is waiting for input, before the main

program has started. Pascal offers no facility for reading, then writing, the same file, or for opening a file with append access. Pascal files are viewed as a sequence of values of one fixed data type, but there is no facility for moving directly to any position in the sequence.

Pascal tries hard to shield the programmer from knowledge of the underlying implementation. This knowledge may not be relevant for understanding what the program does, but it can be critical for efficiency, or for access to externally-defined objects. For example, TEX deals with 8-bit, 16-bit, 24-bit and 32-bit integer values (among others), both signed and unsigned. Pascal's **packed** attribute requests the compiler to pack values more densely, so that one could fit four 8-bit values into a 32-bit word, but the compiler is free to ignore this attribute, and some actually do. Since TEX is a large program to begin with, this one misfeature can make such a compiler useless for implementing TEX. Since Pascal has no provision for distinguishing between signed and unsigned values, a TEX implementor may have to go to some trouble to deal with this; a type declaration like **eight_bits** = 0 .. 255 might result in using more than 8 bits of storage if the compiler thinks that 8-bit values can only have the range -128 .. 127.

Pascal provides no notation for integer constants in bases other than 10, or for bit shifting and masking operations. It does have a **set** type with union and intersection operations, which could be used to implement a bit set; the problem is that the maximal size of the set may be constrained by the compiler, or it may be inefficiently implemented, such as by a list representation. Web provides a translation for octal and hexadecimal constants, but shifting and masking must be simulated by arithmetic operations, paying particular attention to the fact that the underlying arithmetic may be two's-complement, one's-complement, or signed-magnitude. Don Knuth is clever enough to have dealt with this issue correctly, but it certainly does increase the complexity of the programming.

Pascal is strongly typed, which means that combination of values of different types in expressions may be forbidden, or at least require explicit conversion functions to be applied. This typing leads to fewer programming errors, but unfortunately, Pascal goes a step further in making an object's size part of its type. In particular, this means that arrays of different sizes are not conformable, so that if you were to write a function capable of operating on 3×3 matrices, you will need to write another one for the 4×4 case, still another for the 5×5, and so on. Since Pascal character strings are treated as vectors of characters, this restriction makes it illegal to assign an n-character constant string to anything but a variable which has been declared to hold exactly n characters. This makes programming text processing applications in Pascal excruciatingly painful. Even choosing a fixed size for strings is not helpful, if one later has to change it and is then faced with going through the program and adjusting the size of all string constants. Once again, Web comes to the rescue, and introduces a string pool that permits the programmer to use strings of whatever size are needed.

While Pascal does provide for dynamic memory allocation through the new() and dispose() functions, an implementation is free to ignore the dispose() request, and some do. Because of this, TEX and METAFONT are forced to manage their own dynamic memory allocation, and limits on the amount of memory available for this purpose have to be set at compile time. Since several years of TEX experience has shown that these limits were initially rather tight, implementors have generally increased them in their change files, and this of course requires recompilation.

Finally, Pascal makes no provision for separate compilation, which makes the development cycle for large programs much slower than it needs to be, and also makes it impossible to provide libraries of commonly-used functions.

As E. W. Sewell noted, there are Pascal implementations that remove some of these restrictions, but they all tend to do so differently.

DVI drivers have to deal extensively with shifting and masking, with random access I/O, and with large numbers of files, and since their output is how the typeset document is communicated to the output device, they must have precise control over exactly which bits go where, and what the output byte order is. Pascal is just too feeble in these areas.

If one agrees with my view that common source files should support many different operating system implementations and output devices, then the lack of any facility in either Pascal or Web for sharing code via source file inclusion requests is a serious drawback. Some sort of preprocessor facility would be necessary to provide this; while not at all difficult to do, it *is* yet another step that must be done before each compilation. Although much of the source code is the same for each driver, there are minor variants that are desirable or necessary, either because of device peculiarities, operating system requirements, or simply compiler bugs. If this common code is not maintained in a single place, with some sort of embedded conditional processing commands, then it has to be replicated many times over. With the number of devices and environments currently supported by my driver family, that is already a number over one hundred! Web provides support for only one change file, so using it on a master source file when so many variants are required would either require large numbers of similar change files, or use of Web's limited macro capability to create conditional variant sections in the change files.

These criticisms may seem harsh to Web or Pascal devotees. I think Web is a marvelous tool for writing TEX and METAFONT, but it is most definitely *not* a panacea for the maintenance of large software systems.

Well, if Web and Pascal are not suitable for the driver family, what is? Fortran† [ANSI66, ANSI78, ANSI87] has been around for over three decades, and has been implemented on pretty much every commercial computer. Not

† Draft ANSI Fortran 8X recommends the spelling "Fortran", instead of the "FORTRAN" of earlier standards.

only that, Fortran compilers generally produce more efficient object code than do those for any other language; the primary reason for this is just Fortran's long head start and limited number of data types, not for any particularly redeeming features of the language. Fortran has been used for a few DVI drivers, but if the goal of the effort is portability, as it is for me, it is completely unsuitable. It lacks standard bit operations, recursion, source file inclusion, conditional preprocessing, and because it has a record-oriented notion of files, random byte access I/O must be simulated by random record access. For binary files, the records are *Fortran* records; they must be recognizable objects which can be accessed in either forward or backward directions. This in practice is handled by the embedding of magical control fields before and after each record. Fortran records are also word oriented, since the smallest data object in the language is a word containing an integer, real, or logical value. So there goes any chance of getting an uncontaminated byte stream to an output device.

How about Modula-2? Wirth developed Modula (1977), then Modula-2 (1982) [WIRT83], and learned much from the Pascal experience. Modula-2 removes most of the deficiencies of Pascal that I complained about above, but it still leaves some oddities, such as the fact that set size is limited by the host integer word size, which might be as small as 16 bits. Unlike Pascal, which has I/O statements as integral parts of the language, Modula-2 leaves them for definition by library functions. Unfortunately, it does not have a standard run-time library, leading once again to a Babel of implementation variants. Modula-2 has not been as widely implemented as Pascal, and there is a significant share of the computer market for which Modula-2 compilers are simply unavailable.

Ada [ANSI83a] looks good, or at least powerful. It is a very large language designed by committee, with the intent of replacing *all* other languages (including assembly languages) used by the U. S. Department of Defense for "embedded systems", and recently, DoD contract requirements have tended to specify it for other purposes as well. There is a large European commercial interest in Ada, and the language does have the necessary hooks to allow control over object representation, both in the computer, and in an external file. It does not have bit shifting or masking operations, but these could be provided since Ada permits separate compilation and some limited use of routines written in other languages. There is a sufficient collection of inquiry facilities, so that one can write code, particularly arithmetic code, more portably than is possible in any other language. Ada is strongly typed, but array sizes are not considered part of the type. I/O is relegated to library functions, but that library specification is part of the language definition. Random byte access in files is part of that standard. Finally, the DoD Ada compiler validation requirement, and the strict rules against Ada subsetting or supersetting, should greatly enhance portability.

There are two practical problems with Ada, however. First, the size of the language, and the strict validation requirements, have resulted in significantly increased Ada compiler and Ada environment development costs which are passed

on to the end user in fees that are generally several times higher than those levied for compilers for the more traditional languages. Second, the size makes implementations on personal computers, or machines with limited address space, quite difficult. Both of these mean that software development efforts will not have as wide a reach with Ada at present as they could with a different language choice.

Common LISP [STEE84] is a relatively new language with roots almost as old as Fortran. It has a rich run-time library supporting decent file handling, I/O (including random byte access), bit shifting and masking, and has a powerful operating system interface. Like all LISP's, it provides dynamic memory management transparently to the programmer, and like most new LISP's, it can be compiled as well as interpreted, so run-time efficiency might not be an issue. LISP's extensibility has led to almost as many dialects as there are LISP programmers, but Common LISP may bring the community back together. The major problems with it at present are the same as those for Ada and Modula-2—expense and size, or unavailability on many major machines. Despite these drawbacks, I think it would be fun to write a TEX DVI driver in Common LISP.

Thus, by a process of elimination of major candidates (and all the minor ones, because of portability constraints), we are led to the last choice on the list—C. The C language was created by Dennis Ritchie about 1972, based on an earlier typeless prototype language called B, which in turn came from BCPL, also typeless. BCPL's origins are in systems programming and compiler writing, where it is necessary to have close and controllable access to memory bit patterns and hardware registers.

C is used extensively in the Unix operating system, both for the operating system kernel, and for the great majority of compilers and other software tools. It is of sufficient power that it has displaced all but a few hundred lines of assembly code in the Unix kernel. This is a remarkable achievement, probably equaled only by the S-Algol language used to write Burroughs operating systems.

However, C has received a considerable amount of bad press; Philippe Kahn, the founder of Borland International, which markets the popular Turbo Pascal system, has called C a *write-only language*. For readers unfamiliar with C, some explanation is in order.

C is superficially a fairly simple language, not all that different from Algol descendants. Compound statements are delimited by curly braces instead of **begin**/**end** keywords, which is actually an advantage in a text editor that provides for brace matching. C is fairly unusual in that it is case sensitive—TEX, TeX, Tex, and tex are all different identifiers. C has no nested procedures, and makes no distinction between routines that return values (Pascal **function** 's), and those that do not (Pascal **procedure** 's); it calls them all functions. The source file forms part of the variable scoping mechanism—values declared outside a function body are normally globally known, but by giving them the **static** type attribute, their visibility can be restricted to just those functions in that source file.

C has a rich collection of data types. Integer types are **char**, **short**, **int**, and

long, all of which can be signed or unsigned, and floating-point types are float and double. These can be grouped into arrays, unions (like Fortran EQUIVALENCE), and structures (like Pascal records), but without the constraints of Pascal's strong typing on array sizes.

There is an equally rich set of operators to work on data, including shifting, masking, and pre- and post-decrementing and incrementing. However, there are also 16 levels of operator precedence to confuse the programmer.

Pointers can be declared for any data object, and library routines support allocating and freeing memory dynamically. C is unusual in that pointers and arrays are equivalent. Arrays always have a zero index origin, and a[k], *a+k, *(a+k), and k+*a all refer to the $(k+1)^{st}$ element of the array. The asterisk is the indirection, or pointer dereferencing, operator. This feature means that programmers tend to encourage efficient code generation by manipulating pointers, rather than making array references. A simple example of this is a string copy function, which could be written as

```
copy(target,source)
char target[ ];
char source[ ]
{
   int k;
   while (source[k] != '\0')
   {
      target[k] = source[k];
      k = k + 1;
   }
}
```

but would more likely be written as

```
copy(target,source)
register char* target;
register char* source;
{
   while (*source)
      *target++ = *source++;
}
```

These illustrate the pointer/array equivalence, as well as the facts that strings in C are terminated by an ASCII NUL character (represented in C by '\0'), that arguments are normally passed by value (allowing local modification without affecting the caller), and that the post-increment operator, ++, binds more tightly than the indirection operator, *. In the assignment *target++ = *source++, the pointer value represented by source is saved, dereferenced to find the element it points to, then the pointer value is incremented to point to the next element in

the array. The same thing happens for target on the left-hand side of the assignment. Because pointers point to objects of definite types, pointer arithmetic is always defined to move over objects, rather than machine memory locations, so the programmer is protected from having to be aware of the underlying object representation size, which is almost certainly machine-dependent. The register attribute in a declaration is a strong hint to the compiler that the variable should be maintained in a fast hardware register, instead of being materialized in memory.

Extensive type coercion is possible through the use of type casts; for example, on a machine with two's complement arithmetic, (unsigned)−1 says "take the bit pattern represented by the number −1, then treat it as the unsigned integer which has the same bit pattern".

I/O is not part of the language, but is provided through library routines which have followed the original Unix file system and I/O model in most current C implementations. In particular, files are simply streams of ASCII bytes which end at the last byte written, and have no other structure imposed on them. There are consequently no concepts of records, blocks, carriage control options, or text versus binary.

C therefore has a suitable set of primitives, data types, and library routines to provide the tools we need to write a TEX DVI driver. In addition, it has a preprocessor which permits definition of simple macros, optionally with arguments:

```
#define MAXPAGEFONTS 16
#define DEVICE_ID "Hewlett-Packard LaserJet Plus laser printer"
#define OUT16(n) {OUTC((n)<<8); OUTC(n);}
```

and which provides for conditional preprocessing based on preprocessor expressions:

```
#if (CANON_A2 | HPJETPLUS | IMPRESS | POSTSCRIPT)
  (void)bopact();
#else
  (void)clrbmap();
#endif
```

Finally, the preprocessor provides for source file inclusion:

```
#include <stdio.h>
#include "prtpage.h"
```

The angle-bracketed form is used for system-defined include files, and the quoted form for user-defined ones.

In summary, C has all the tools we need, and implementations are available on virtually every commercial machine from personal computers up to supercomputers. It is by no means restricted to byte-addressable machines; on our

DEC-20, which is a 36-bit word-addressed machine, there are at least 5 different C compilers available.

As Kahn's comment above indicates, C does have pitfalls:

- The many levels of operator precedence and excessive use of pointers are highly prone to programming errors, as is the writing of overly-complex expressions using C's rich operator repertoire.
- As in Algol 60 and 68, assignments are legal expressions which return the value of the left-hand side. This leads to a potential error when the assignment operator, =, is confused with the equality test operator, ==. The programmer who writes the legal C statement if (a = b) ... probably does not mean to assign b to a, then test whether a is non-zero, but instead, wants an equality test, if (a == b) .... A few compilers issue a warning message when they see the former construction, but the error is still quite common, and quite hard to spot.
- Array objects do not carry their sizes around with them, and frequently masquerade as pointers, so the compiler cannot generate bounds-checking code. Many of the run-time library routines return values in array arguments whose sizes are assumed to be "big enough", but are otherwise unavailable to the routine. That is, of course, a library design flaw, not a language defect.
- There are tiny syntactic peculiarities that can lead to hard-to-find errors. Examples include a semicolon following a **while** or **for** loop condition, which gives the loop a null body, instead of having it act upon the following statement, and the fact that execution of a **case** in a **switch** statement by default falls through to the next **case**, instead of exiting the statement. The **break** statement provides the necessary **case** exit, but it is easy to forget.
- Because early C compilers were implemented on machines which had a uniform byte-addressed memory architecture, where the memory address filled one integer word, many programmers assumed size and type equivalence of pointers and integers, and assumed that the constant 0 was equivalent to C's null pointer value. This does not hold on other architectures; the most notable exception is the Intel iAPX architecture used in the IBM PC, where "far" pointers are generally composite objects containing a segment descriptor and a segment offset, and which in fact fill two integer words of storage. On word-addressed machines which support packing multiple characters in an integer word, such as the DEC-20, character pointers have an entirely different format from pointers to other data types, although they may still fit in one integer word.
- Again, because early C compilers were implemented on similar architectures, the DEC PDP-11 and VAX computers, programmers made assumptions about the absolute sizes of the integer data types, **char**, **short**, **int**, and **long**, and about the memory storage order of bytes, which are likely to be incorrect for other architectures.

- C provides for separate and independent compilation of functions. Until the draft ANSI C Standard [ANSI86] was prepared in 1986, there was no way for the programmer to tell the compiler the number and types of arguments passed to a function, so no module interface type checking was possible. This is a recognized deficiency, and draft ANSI C (which may be adopted in late 1988) introduces a new function prototype declaration to provide for this. Several compilers have already implemented support for these prototypes, uncovering innumerable long-lived user-program bugs in the process.
- The fact that arrays in C have a zero origin, so that an N-element array is indexed A[0] ... A[N-1], leads to frequent "off-by-one" errors in array references, since most humans start their counting at 1, not 0.
- There is no provision for declaration of variables having a subrange of integer values. Pascal, Modula-2, and Ada provide this, and will enforce it either at compile time, or at run time. In C, one can only declare it generically as one of the four standard integer types, with an optional sign attribute. There is no run-time range checking, and integer overflow is generally ignored.

Host Environments

The DVI driver family has been carefully designed to be portable across multiple operating systems and different host architectures. Of course, some system dependencies are inevitable, but they can be nicely handled through C preprocessor conditionals. Here are the systems currently supported at version 2.10, with compilers noted in brackets:

- Atari 520ST+ GEMDOS [Mark Williams C];
- MS DOS (IBM PC family) [Microsoft C];
- TOPS-20 [PCC-20 and KCC-20];
- VAX VMS [VMS C];
- Unix (most variants, including BSD and System V).

At the time of writing, ports are under way to IBM CMS [Waterloo C], Prime Primos, and Data General, but are still incomplete. There may be others that I have not yet heard of.

Output Devices

Here is a list of the currently-supported output devices:

dvialw	PostScript (Apple LaserWriter and others)
dvibit	Version 3.10 BBN BitGraph terminal

dvican	Canon LBP-8 A2 laser printer
dvie72	Epson 9-pin 60h × 72v dpi dot-matrix printer
dvieps	Epson 9-pin 240h × 216v dpi dot-matrix printer
dvigd	Golden Dawn Golden Laser 100 laser printer
dviimp	Imagen imPRESS-language laser printer family
dvijep	Hewlett-Packard LaserJet Plus laser printer
dvijet	Hewlett-Packard LaserJet laser printer (really a 100 dpi dot-matrix printer)
dvil3p	DEC LN03 Plus 150dpi and 300dpi laser printer
dvil75	DEC LA75 144 dpi dot-matrix printer
dvim72	Apple Imagewriter 72 dpi dot-matrix printer
dvimac	Apple Imagewriter 144 dpi dot-matrix printer
dvimpi	MPI Sprinter 72 dpi dot-matrix printer
dvio72	OKIDATA Pacemark 2410 72 dpi dot-matrix printer
dvioki	OKIDATA Pacemark 2410 144 dpi dot-matrix printer
dviprx	Printronix 60h × 72v dpi dot-matrix printer
dvitos	Toshiba P-1351 180 dpi dot-matrix printer

There are also a couple of experimental drivers which are not listed here.

Adding support for a new dot-matrix printer is quite straightforward, starting from the code for a similar printer. A few size parameters in the file header need to be changed to define the resolution and paper size, and a single output function (50 to 100 lines of code) needs to be rewritten to convert the bitmap in memory (stored in scan-line order) to the sometimes bizarre format needed by the printer.

Support for new laser printers or screen displays involves substantially more effort, since it involves downloading fonts to the device, which may have its own peculiar conventions, such as having forbidden regions in the ASCII code sequence where fonts may be defined, or putting restrictions on the number of fonts, or size of characters, that can be defined.

I have received promises of adaptations to several other printers, but none have arrived at the time of writing. Doug Henderson (UC Berkeley) has reported some work on the Mergenthaler Linotronic 300 phototypesetter, which is a high-resolution PostScript printer, but the project is currently suspended due to lack of funding.

Support for TEX \special Commands

The TEX \special command is the mechanism by which a TEX document can get an arbitrary string into the DVI file. The intention of this is to provide for

facilities that TEX itself does not offer, but which the DVI driver could supply. Here are some typical examples:

- verbatim insertion of graphics files;
- invocation of device graphics commands;
- invocation of forms overlays (e.g. institutional letterheads);
- selection of alternate paper trays;
- temporary switch to landscape printing;
- requests for manual paper feed;
- changing foreground and background colors on a color display;
- gray-shading a rectangular region; and
- printer operator instructions.

Use of a \special command potentially inserts a *device* and *driver dependence* in TEX's device-independent output, and is therefore a barrier to document portability. This is a recognized problem, and a committee of the TEX Users Group has been formed to come up with recommendations for the provision of better, and more uniform, support for \special commands.

In the meantime, this DVI driver family provides \special support only in the PostScript driver, dvialw, and then only in a fairly simple form. It is nevertheless possible to edit PostScript files produced by other programs into the form recognized by dvialw, and then have a \special command request their insertion, either at the current point on the page, or overlaid on the entire page.

I have received contributed code for the Hewlett-Packard LaserJet Plus driver that provides similar support for dvijep, but I have delayed installing it until the recommendations of the above committee have been published. Since I am supporting a *family* of drivers, it is important that features provided by the \special command be available in all of them, unless it can be demonstrated that the need for a facility available only on a few devices is so great that it is worth adding anyway.

There is only one single function, special(), to be rewritten in the DVI driver code. With the exception of dvialw, this is just a dummy routine that prints a message noting that it has been called. The version of special() in dvialw is about 200 lines of code.

Users who are desperate for such support have been able to add it in their local copies without great difficulty, and on request, I have distributed copies of the dvijep code extensions.

Driver Installation

The driver code consists of 20 source files, dvixxx.c, where xxx is the mnemonic for the output device, plus about 70 .h files which are #include'd by them. Each driver file includes between 50 and 60 of these files.

The reason separate compilation is not used for the included files is that most of them contain preprocessor conditionals which select different code sections, depending on the output device, host operating system and compiler, and so on. If they were separately compiled, it would be impossible to keep track of which code sections were enabled in any particular compiled object file, and chaos would ensue if a driver were loaded with the wrong versions. Presenting a single source file to the compiler removes this problem, though it does suffer from the same objection I raised earlier about Pascal, namely, recompilation of the entire file is wasteful and slow.

On an IBM PC XT (4.77MHz Intel 8088) with Microsoft C Version 4.0, compilation and linking of a single driver takes about 20 minutes; Version 5.0 of the compiler can now be used with optimization, but that increases the compilation time to 30 minutes. On a Sun 3/280 (25MHz Motorola 68020) Unix file server, it takes 35 seconds.

Provided the host system is one of those for which support has already been implemented, all that is necessary after loading the source files onto the system is to edit one single file, machdefs.h, and follow the instructions in its leading comments to make the necessary local file-naming changes, and select the appropriate operating system and compiler settings. Since the latter are already there nested inside comments, in practice, all I need to do with the code when I move between local machines is move two bracketing comment lines.

Unix-style makefile's are provided for each of the operating systems, and all that is needed is to copy the appropriate machine-specific one (e.g makefile.vms) to a file named makefile, then type make to initiate the compilation. A public-domain implementation of make that runs under all the supported operating systems is provided with the DVI distribution, and should be used for this purpose. Doing it any other way is guaranteed to be the hard way, and will almost certainly introduce errors.

For some systems, a couple of extra separate files may be compiled and loaded with the driver file, but this is handled automatically by the commands in the makefile.

How Big is a Driver?

You might be curious to know how much code it takes to write a DVI driver program. At the present time, the shared code amounts to about 20K lines of C in 39 .c files and 69 .h files. The size of a single driver varies from 7800 to 9000 lines, counting lines in the .h files it includes; after stripping comments and blank lines, this reduces to 3300 to 4000 lines of actual code. With 20 drivers, family code sharing saves $20 \times 8000 - 20000 = 140000$ lines; that number represents how much extra code would have to be maintained if these drivers were not treated

as a family.

Documentation consists of a 94-page installation guide (4600 lines), and a 20-page user manual (1100 lines), both in LaTeX form. The user manual is also available in TeXinfo form for TOPS-20 and GNU Emacs INFO on-line access, and as a Unix nroff/troff manual "page", for hardcopy and on-line access.

It is interesting to compare the driver sizes with some other TeXware. The Pascal output of Tangle is 80-character packed line images with no comments or embedded space, and usually several statements per line. To make these comparisons, the code was run through the Berkeley Unix Pascal prettyprinter, pxp -O -f, so that it looks more like it was hand-written.

Program	Lines
dvitype	1700
gftype	900
pktype	550
METAFONT	20170
TeX	20119
Common TeX	19600
CTeX	22760

It is remarkable that the TeX and METAFONT Pascal programs are within 51 lines (0.2%) of one another. The two C translations of TeX are also very close in size to the Pascal version.

Portability Considerations

Software portability is not a subject commonly covered in books. For the C language, there are only three books which I have found useful.

Harbison and Steele [HARB87] is currently the best source of a description of the complete language; the second edition includes features introduced by draft ANSI C [ANSI86].

Lapin [LAPI87] (a pseudonym for the staff of Rabbit Software) has a good discussion of portability issues, and provides useful tables of library function availability under six major versions of the Unix operating system.

Rochkind [ROCH85] is a solid reference for Unix systems programming applications. It was invaluable for preparing the code needed to support immediate keyboard input for the BBN BitGraph DVI driver, dvibit.

The original definition of C by Kernighan and Ritchie [KERN78] is written in tutorial form, rather than as a reference manual, and as the first book on the language, has been followed by most implementors. However, certain parts of the

language are not clearly defined, and the run-time library is only briefly covered. Also, concern about possible infringements on AT&T software copyrights has led some vendors to provide completely different run-time libraries.

I believe it is worthwhile to summarize here some of the main considerations, because until these are thoroughly understood, digested, and learned, C programmers have little chance of writing portable code. These are stated in the form of rules:

- external names unique in first *six* characters;
- no mixed-case function or variable names;
- file names unique in first *six* characters, and limited to *eight* characters, plus *three* character extension;
- file names in one case;
- preprocessor names unique in first *eight* characters;
- consistent, readable, code formatting, e.g.

```
#define BAR 'b'
#define FEE 'f'
void
foo(c)
char c;
{
    if (c == BAR)
        (void)bar();
    else if (c == FEE)
        (void)fee();
    else
        (void)def();
    for (k = 1; k < n; ++k)
    {
        (void)printf("Iteration %d",k);
        NEWLINE(stdout);
    }
}
```

- source code lines limited to 80 characters (C has backslash-newline convention for continued lines);
- type declarations declare one variable each, and have a descriptive comment:

```
char curpath[MAXFNAME]; /* current file area */
```

- type declarations ordered alphabetically;
- no labels or *goto* statements;
- explicit type casting of expressions and assignment in mixed-mode is *required*;
- no binary I/O on stdin/stdout;

- NEWLINE(file) instead of printing newline character;
- functions which do not return a value are typed void;
- discarded function values are type cast as (void);
- all functions declared before use, with ANSI and non-ANSI prototypes:

```
#if   ANSI
void  actfact(UNSIGN32);
#else /* NOT ANSI */
void  actfact();
#endif /* ANSI */
```

- main() function defined first in source file, with all others following in *alphabetical* order;
- explicit integer/Boolean typing using typedef's for BOOLEAN, BYTE, COORDINATE, INT8, INT16, INT32, UNSIGN8, UNSIGN16, and UNSIGN32;
- parenthesize expressions—C has 16 operator precedence levels;
- null pointers are (type *NULL), not 0; pointers are not equivalent to integers;
- no use of enum, functions returning struct, or non-library functions with variable numbers of arguments;
- no preprocessor symbol concatenation possible (i.e., do not use #define A(x) A/* */x or #define A(x) A##x);
- preprocessor symbols entirely in UPPER-CASE;
- preprocessor #if expressions limited to simple variable, or (VAR1 | VAR2 | ... | VARn);
- cannot assume availability of compile-time define capability for preprocessor symbols (e.g. cc -DOS_UNIX -c foo.c);
- no white space *before* or *after* # in preprocessor statements;
- preprocessor statements allowed: #define, #if, #else, #endif. Restricted use of #undef, #ifdef, #ifndef. No #elif, #error, #if defined(var), or #pragma.

When code is contributed to the DVI driver family, I first go through it manually to check for any violations of these rules, and make repairs as necessary. Then the code is processed by compilers on several operating systems with full error checking enabled, and the Unix lint utility is run on it. Any problems these reveal are fixed, and the compilations and lint runs are repeated. Once this has been done, I can be reasonably sure that ports to new machines should be considerably easier.

It is also a requirement that a feature may not be added that is peculiar to one compiler or operating system, or to one output device, without strong justification. An example of such an exception is the option for automatic spooling of the driver output, which is available on TOPS-20 and 4.2 (or later) BSD Unix; it is just too useful to leave out. I have not been quick to implement special requests for things like landscape mode printing in one particular driver. That is a general capability which they all deserve, but when it is done, it must be done with sufficient generality.

Environment-Induced Code Modifications

I noted earlier that parts of TEX and METAFONT are necessarily system-dependent, and that it is necessary to provide modifications for these sections with a change file. Here are the sizes of the TEX Web change files for selected operating systems:

Operating System	Lines
IBM CMS	1850
IBM MVS	660
TOPS-10	1030
TOPS-20	1300
Unix	1360
VAX VMS	1080

In the DVI driver family, such changes are handled by C preprocessor conditionals. At last count, there were 131 operating-system sections, and 293 output-device sections. Although these numbers are not small, neither are they big compared to the 20000 lines of code involved.

I was curious to see just what the distribution of these conditional code sections is, so I ran a minor modification of D. McIlroy's Unix implementation of the word frequency program given in [BENT86], and obtained the following table. Names with fewer than 5 references are dropped. It is common for such code sections to apply to more than one device or operating system, so the frequency totals in this table exceed the number of conditional code sections.

Code Section	Frequency
SEGMEM	71
HIRES	44
IBM_PC_MICROSOFT	43
POSTSCRIPT	43
CANON_A2	40
HPJETPLUS	40
BBNBITGRAPH	37
IMPRESS	28
OS_VAXVMS	27
OS_TOPS20	26
IBM_PC_LATTICE	20
PS_SHORTLINES	19
IBM_PC_WIZARD	16
OS_UNIX	16
OS_ATARI	14
PCC_20	11
VIRTUAL_FONTS	10
KCC_20	9
BSD42	7
EPSON	7
HPLASERJET	7
OS_PCDOS	7
HOST_WORD_SIZE	6
STDRES	6
APPLEIMAGEWRITER	5
GOLDENDAWNGL100	5

The operating system, compiler, and output device names should be fairly obvious. The problems I have had with the IBM PC are reflected in the larger numbers in this table. SEGMEM refers to the effects of the Intel iAPX segmented memory, to be described later. HIRES sections are from those device drivers (e.g. dvioki and dvio72) that have both high and low resolution output modes; code for the latter occurs in the #else branch of a preprocessor conditional. PS_SHORTLINES code sections limit the PostScript output line width. VIRTUAL_FONTS code sections implement one-time reading of font files to improve performance across networks.

Output Device Misfeatures

Dealing with output device misfeatures can be a frustrating experience. The first problem one faces is device documentation that is unclear, inaccurate, incomplete, or just plain wrong. Every laser printer I have used has suffered from this problem.

Apple's initial documentation of downloading fonts into the LaserWriter was simply incorrect; it took a visit to Adobe Systems to find out how to do it. Apple ships its US$7K LaserWriter printer with a manual that tells little more than how to unpack the printer from its shipping box and install the paper and toner cartridge, but nothing whatever on what the RS-232C serial connection pin assignments are, and nothing on programming it. For information about such things, you are expected to order a separate technical reference manual.

Hewlett-Packard's LaserJet Plus documentation of font downloading was unclear, but fortunately, their telephone support is excellent, and it took only one weekend's work to get an initial driver going. Now for their LaserJet Series II printer, however, the documentation is truly outstanding; they just don't pack the Technical Reference Manual with the printer like they did with the LaserJet Plus, and you need that manual to program the printer. Once again, their support is good, and I got a manual expressed to me the same day as my phone call to them.

The Canon LBP-8 A2 comes with a drastically abbreviated manual, where in the font downloading section, you are instructed to send character "sizes" (width? height? depth? units?) followed by "binary data" (what binary data? and how much?). The local dealer had tried unsuccessfully to obtain the more detailed Subsystem Manual, and I only got a copy from Canon U. S. headquarters on the grounds that I was a software developer that had a product that used their printer.

> **Adequate documentation, both for end users and for programmers, ought to be considered an inseparable part of a hardware product costing thousands of dollars.**

A number of printers are now on the market which claim to emulate other popular printers. The developers of these products are faced with much the same problem a DVI developer has, only they must be able to prepare a *complete* software emulation based on whatever documentation they can obtain. Needless to say, they do not always succeed. The Mitek and Personal Computer Products LaserImage 2000 printers, which emulate the Hewlett-Packard LaserJet Plus, among others, are known to work correctly with dvijep, but the DataProducts LZR, Kyocera, QMS 800+, and Texas Instruments Omnilaser printers do not. Kyocera has admitted the faulty emulation, and will be issuing new ROM's; I trust the others will follow suit.

Dot-matrix printers seem to be designed for the convenience of the programmer who wrote their ROM code, rather than the thousands of users who must

program them. With the exception of the Printronix and Hewlett-Packard LaserJet printers, all that I have encountered use a raster image encoding based on sending vertical bit clusters corresponding to the number of pins in the printing element head (usually, 6, 7, 9, 16, or 24). This is more complex to encode, and also reduces the possibility of run-length encoding giving significant data compression. The Printronix accepts data in scan-line order, but requires the bits encoded in *reverse* order in each 6-bit group! What is more, some of these printers forbid the printing of two adjacent dots on a raster line, on the threat of "burning out the print head".

This is just plain nonsense. It should be up to the printer to re-encode raster bits presented in scan line order into whatever format is needed internally to fire print hammer pins, and to prevent printing adjacent dots, if this is indeed a problem.

Some dot-matrix printers offer a higher density print mode made by multiple passes of the print head across the page. Unfortunately, the carriage positioning precision is lacking, and the print quality may be worse than that of the one-pass low-resolution mode.

Several dot-matrix printers offer run-length encoding (i.e., sending a repeat count followed by a bit pattern when that pattern is repeated in the raster image). This reduces the data volume that must be sent to the printer, and could do an even better job if horizontal scan line encoding were used, but in almost every case where I have tried it, the printer either runs slower than it would for the full bitmap, or does not print correctly. Consequently, run-length encoding is left as a run-time option for those DVI drivers.

With the exception of PostScript and Imagen's imPRESS, none of the output devices I have encountered provide for the insertion of comment text in a file. Such text can actually be quite useful, even if the printer ignores it, since it can be used to record archival information about the file, such as author, creation date, filename, and host origin. Although a limited amount of such information may be available in the file attributes on the machine on which the file was created, there is rarely any provision for the creating program to add its own remarks. Anyway, the attributes are generally lost or damaged when the file is transferred to another machine, and that occurrence is becoming increasingly common in an era of networks and distributed computing systems. Binary files are unintelligible enough for a human without making it even harder by outlawing embedded comment strings.

There are a number of terminals on the market that support downloaded fonts, but their restriction to fixed-width fonts makes them utterly useless for TEX display. Considering that memory prices are now so low that the 128Kb of RAM chips for a 1024 × 1024 display cost less than US$15, and an extra 128Kb or 256Kb would be ample for font storage, this feature lack is deplorable. The BBN BitGraph remains the only terminal I have found so far which is adequate for TEX previewing. It has some oddities too. It allows only 3 fonts (which is

not many, but one can live with it), but only provides for the definition of 96 characters in a font, so each 128- or 256-character TEX font must be mapped into more than one BitGraph font.

The Hewlett-Packard LaserJet Plus and Series II printers define fonts with 16-bit parameter sizes ($-32768 \ldots +32767$, or $0 \ldots 65536$), but limit the number of fonts per document to 32, and the number of fonts per page to 16. There seems to be no obvious reason why this second number should be smaller, unless some programmer saved a byte of memory by squeezing a font number into 4 bits somewhere else.

Since the two 32-character control character columns in 8-bit ASCII are excluded, downloaded character numbers are limited to the ranges $32 \ldots 127$ and $160 \ldots 255$. This requires a remapping of TEX character numbers, which can be hidden in a short preprocessor macro.

The Plus limits character extents to the range $-127 \ldots +127$ which is too small for characters larger than 36pt, or even for smaller ones, if they have large descenders or ascenders. The Series II raises that limit to the peculiar range $-4200 \ldots +4200$, whose limits are not even powers of two; perhaps they reflect the number of dots down a page of U. S. legal-size paper. The width and height are required to be at least one bit, so that one cannot define a zero-width character.

The LaserJet Plus and Series II have another oddity. There is a command to delete a downloaded font, but it is of little utility, because it causes an immediate printer page eject!

Version 2.10 of the DVI driver family incorporates a major rewrite of **dvijep** which removes both the document and page font count limitations, as well as the character size limitations; when necessary, **dvijep** will now revert to sending characters as bitmaps, instead of as downloaded fonts.

PostScript [ADOB85] printers have lots of problems too. The Apple LaserWriter with Version 23.0 PostScript ROM's has a couple of annoying bugs. With serial RS-232C communication, it can send an X-OFF character to suspend transmission from the host, then forget (under circumstances which are hard to reproduce, but still occur frequently) to send an X-ON character to resume transmission. Eventually, it times out and flushes the rest of the job. It also gets random virtual memory errors; the same job may print successfully twice, and fail a third time. Adobe eventually produced a fix for the X-ON/X-OFF problems, and the Version 38.0 ROM's shipped with the Apple LaserWriter Plus removed the bug. As of September 1987, new Pluses are being shipped with Version 47.0 ROM's, and existing Pluses can be upgraded to these. Besides bug fixes, these new versions have introduced significant performance improvements in the PostScript interpreter. With the Plus (Version 38.0 ROM's), I have been able to obtain 5 pages/minute output (the print engine is capable of 8) on a 100-page document, whereas the old LaserWriter (Version 23.0 ROM's) barely averaged 2 pages/minute.

There are some other problems with current PostScript implementations. One is that performance is severely hampered by the lack of floating-point hardware; computations in PostScript are carried out in IEEE floating-point format (which is commendable), but most printers use software emulation. Timing loops in the Apple LaserWriter reveal speeds of about one millisecond for an add, subtract, multiply, or divide, which is about 100 times slower than hardware floating-point would provide.

A second problem is that there is insufficient virtual memory available. The Apple LaserWriter is probably the worst offender, with only about 150Kb remaining at job start time; some other vendors have 400Kb or more available. This is unfortunate, because with chip prices below US$100 per megabyte,† there should be lots of memory in the printer. This is less critical for jobs that use device-resident fonts, but for downloaded font applications, like TEX with Computer Modern fonts, it is a serious limitation.

A third problem with PostScript is that it is verbose, and no compact binary form of PostScript exists, as it does for Imagen's DDL language. The PostScript produced by dvialw introduces one-letter abbreviations for common command sequences, and eliminates all unnecessary white space, in an effort to reduce the amount of data that must be sent to the printer. The Hewlett-Packard LaserJet Plus and Series II use a binary representation for font definition sequences, but verbose ASCII digit strings for positioning commands.

Here are some comparisons of the DVI driver output file sizes (in bytes) using the TEXbook as a test document. The tests were carried out on a Sun 3/280 (25MHz Motorola 68020). First we process a single page (page 85 in Chapter 13), which will have an initially larger overhead because macros are being defined, and fonts are being downloaded.

Driver	Time (sec)	Time Ratio	File Size	Size Ratio
dvialw	10.2	1.16	37324	2.42
dviimp	8.8	1.00	15452	1.00
dvijep	9.1	1.03	21671	1.40

Next we consider a larger number of pages (Chapters 13–16, 53 pages), which would be more typical, perhaps, of user TEX manuscripts. After several pages have been processed, fonts have already been downloaded, and the output consists primarily of positioning commands, and text to be typeset.

† based on advertised single quantity prices in November 1987; manufacturer's volume prices would be much lower.

Driver	Time (sec)	Time Ratio	File Size	Size Ratio
dvialw	68.0	1.59	753326	1.95
dviimp	42.7	1.00	386554	1.00
dvijep	46.3	1.08	668543	1.73

Note the significant startup overhead for the output of a single page; it took more than 8 sec/page, while the 53-page output averaged about 1 sec/page. Part of this is due to the large size (1.97Mb) of the DVI file, and part to the large number of fonts (54) which must be opened to retrieve font definitions. More modestly-sized documents would not have such a large startup time.

Imagen's imPRESS language as output by dviimp produces the most compact output; it uses 16-bit binary numbers for absolute positioning commands, instead of ASCII digit strings. PostScript output by dvialw requires nearly twice the data that imPRESS needs. The large size ratio for dvijep, compared to that in the preceding table, is partly due to the use of ASCII digit strings for text positioning; towards the end of the output file, positioning commands require on average about 3.3 times as much output as the text to be set. Another contributing factor is that the TEXbook exceeds the HP Laserjet Plus document font limit, so some characters must be sent repeatedly as bitmaps, rather than as downloaded fonts; in these 53 pages, there were only 63 characters that fell into this class, but they contributed 154Kb, or 23%, to the file size. If the font count limitation did not exist, the file size ratio figure would drop from 1.73 to about 1.32.

A fourth problem with PostScript is that insufficient thought seems to have been given to the problem of getting data to the printer. PostScript uses only printable ASCII characters for commands, except for two control characters used for status requests and end-of-job signals; other control characters are just ignored. However, a line break is the same as a space, except in a string to be typeset, where it is part of the string. PostScript defines a backslashed newline in a string to be ignored, but outside of strings, that sequence cannot be used to get an ignorable line break. If backslash newline were uniformly ignored by PostScript, it would be trivial on record-oriented file systems to ignore record boundaries; as it is, a program must take considerable care on such a system to break long PostScript sequences in one of the allowable ways. Version 2.10 of dvialw goes to considerable trouble to provide this facility so that a maximum line length limit can be enforced for those systems that need it.

The reaction of PostScript on a transmission error is to immediately flush to end-of-job; in our experience, such errors are not uncommon. By contrast, Imagen laser printers support a checksummed file-transfer protocol to allow error-free data transmission between host and printer; so far, they seem to be unique in the industry in this respect.

Given that the Kermit [DACR87] file transfer protocol is in the public-domain, and widely-available Kermit implementations exist on almost every commercial machine, including microcomputers, it does not seem unreasonable to ask vendors of intelligent printers to consider adding Kermit file transfer for reliable data communication.

A fifth problem is that current PostScript implementations do not have garbage collection (memory reclamation). When you issue a command like (somestring) show to display a text string, the string continues to occupy memory after it has been printed. Adobe recommends wrapping each page with save/restore commands to reclaim this wasted memory, but if that is done with downloaded fonts on the page, they are lost. Version 2.10 of dvialw incorporates a major overhaul to separate the font downloading on a page from the text set on the page, and the text is then bracketed by save and restore. This has significantly increased the number of pages that can be printed on the Apple LaserWriter before it aborts the job with a "VM error" condition. If 500Kb were available, instead of 150Kb, the problem would likely disappear for most TEX documents.

It should be pointed out that it is *not* the job of the DVI driver to communicate directly with the printer to find out how much memory it has. It should be regarded simply as a filter that transforms one kind of file, a DVI file, to another, a PostScript file. A spooler for a PostScript printer is already a fairly complex program, because the printer sends back all sorts of verbose status messages that must be dealt with. But without dynamic communication with the printer, it is just impossible for the DVI driver to know how much memory is available in the printer, because that memory amount changes dynamically; other jobs can define objects that consume memory and survive beyond end-of-job.

Talaris has provided a solution for this problem with another printer family, the QMS laser printers. They provide a spooler which is also a font manager. It strips out fonts from documents and only sends what it knows the printer does not yet have. This can only work, however, if the printer command language is simple, because the spooler must be able to interpret it completely in order to identify font definitions. Like LISP, PostScript is an extensible language, and the only way to do the analogous thing with it would be with a full-blown PostScript interpreter, which is a *very* large job.

I feel quite strongly that the only real solution to the problem of device font storage is adequate memory in the first place, and it is a relatively economical solution today.

Host Misfeatures

With the exception of 4.2 and 4.3 BSD Unix, the drivers have uncovered problems

on each of the systems they have been implemented on.

The IBM PC, announced to the world on August 12, 1981, has probably the sorriest history of any commercial computer with a long record of positively abysmal bug-ridden compilers and assemblers. Considering that this computer has the largest user base of machines, with an estimated 12 million PC's and clones having been sold by early 1987, this is really tragic.

In my view, part of the problem lies with the complexity of the Intel iAPX design, which has a segmented-memory architecture, and dedicated, rather than general-purpose, registers. The 80836 (IBM PS/2 model 70 and a few clones) has 32-bit registers, but the 8088 (IBM PC and PC XT), 8086 (IBM PS/2 model 30 and many clones), 80186 (some clones), and 80286 (IBM AT and PS/2 models 50 and 60, and many clones) all have 16-bit registers. However, they have a 20-bit address made up from a 16-bit offset and a 16-bit segment register. The segment register contents are either left-shifted 4 bits and added to the offset, or in protected mode on the 80286, index a base address table, the selected entry being added to the offset to obtain the address.

When an array is known to lie within a 64Kb memory segment, computing an array element address can be done with one instruction, as is the case on most current architectures (RISC architectures may take perhaps two or three instructions). However, as soon as the object is larger than 64Kb, a complex sequence of about 30 instructions must be executed to piece the segment and offset together into a 20-bit number contained in a pair of 16-bit registers, where the index can be added using multi-precision arithmetic, and the steps must then be reversed to recover a new segment and offset pair.

Procedure calls in the case of multiple-segment code do not require more instructions than they do in single-segment code, but they *do* push different (and incompatible) return address formats on the stack. The combination of different addressing modes and calling sequences requires vendors to offer up to 6 different versions of each of their libraries.

The iAPX architecture does not define a standard calling and argument-passing sequence like the DEC VAX does, with the result that there are about as many different conventions in use as there are compilers, generally making it impossible for different compilers to be used in the same code development project. There is not even uniform adherence to the Intel standard object file format; some compilers come with linkers that can only be used with object code from that one product. That, plus the diversity of addressing modes and memory models, makes it very difficult for software vendors to provide subroutine libraries only in object form.

Because of the addressing complexity, most early compilers did not support code or data larger than 64Kb, which is a serious limitation for a machine that should be able to address 640Kb of memory. Why not 1Mb if the address is 20 bits ($2^{20} = 1048576$)? That is another story which I won't go into here. Later compilers have added support for multiple memory models, but most are still

rather buggy.

Another problem with the IBM PC is that it lacks memory protection; some clones do not even have memory parity checking. There is no hardware stack overflow and underflow detection, and the stack is limited in size to 64Kb. Since local variables are allocated on the stack in most modern languages, this is a serious limitation on program data size, and often requires rewriting code ported from other systems that are not plagued by such severe stack-size limitations. A software bug, such as an out-of-bounds array reference, or an incorrect number of arguments to a function, is very likely to wipe out some portion of the operating system, or the stack, hanging the machine. It is conceivable that just the right (wrong?) combination of data written into the disk controller's tables could even destroy your hard disk. When the machine hangs, you must cycle the power and reboot, which of course destroys the memory image, and any chance you might have of diagnosing the problem.

It should be noted that the Apple Macintosh has the same problem of lack of memory protection, and lack of stack underflow/overflow detection, although its stack can be considerably larger than 64Kb.

The PC lacks proper handling of interrupts; a looping program cannot be stopped by keyboard input of a Ctl-C or Ctl-Break character, since all that does is set a flag that the operating system checks the next time you call it. If the program is in a compute loop, that will never happen, and your only recourse is the power switch.

Another problem, which I think is common in the microcomputer industry, is the relative youth of many of the individuals involved; having failed to learn history (of mistakes made in the 1950's and 1960's in computer software), they are repeating it. Adequate quality control, even from some major microcomputer software companies, seems to be unknown.

For the DVI driver family work, Lattice C, Aztec C, Wizard C, and Turbo C (which evolved from Wizard C) have all been tried and found deficient. Either they do not compile the sources at all, or they compile but fail at run time.

Microsoft C became usable at Version 4.0, and I thought for a long time that there was only one library bug that affected the drivers, and that fortunately had a simple workaround. However, just this week, we have found another bug—the `fseek()` function is used for random access file positioning, and under reproducible, but rare, circumstances, it can fail to correctly position the file. Since Microsoft C Version 5.0 is due out, in the immortal words of Byte Magazine columnist Jerry Pournelle, "real soon now", I have shelved further attempts to more precisely define the bug.†

† Note added in proof: Microsoft C 5.0 arrived in mid-November, 1987, and has been used for the preparation of the floppy disk distribution of version 2.10 of the DVI family. The two bugs noted above have disappeared, and it now seems possible to run the code when it is compiled with optimization. With

The segmented memory architecture and restricted stack size do affect the drivers. The limited stack size means that care has to be taken to avoid having large local arrays; they must be either allocated dynamically at run time, or made static globals at compile time. The bitmaps used by several of the dot-matrix printer drivers exceed the 64Kb segment limitation, so it is not possible to address them directly; instead, they are dynamically allocated as arrays of pointers to scan lines. The coding complexity is reduced by hiding this ugly indirect access mechanism in macros.

The 16-bit register sizes of the Intel iAPX family (up to the 80286), and the extra code needed to deal with 32-bit integer arithmetic imply a serious speed and code size penalty for programs that use 32-bit arithmetic where a smaller size would do. Consequently, I have been extremely careful in the DVI driver family to create new private 8-, 16-, and 32-bit signed and unsigned integer types, and to use the smallest type possible, instead of the generic types **int** or **long**. The DVI file format clearly defines its integer data value sizes, so it was straightforward, though tedious, to implement this change, and it has since become a programming habit that I adhere to in writing C code.

On the Atari 520ST+, the Mark Williams C compiler proved to have some limitations about the size of input statements which required simplifying them. Similar problems showed up in Sun OS 3 with some expressions of quite modest size, and again, code has had to be rewritten to allow it to compile without error.

PCC-20 on the DEC-20 is a port of Steve Johnson's Portable C compiler which has been used as the initial implementation of C on most Unix systems. It has some annoying bugs and misfeatures which may be present in other PCC ports as well. The preprocessor incorrectly collapses to zero a bit mask which has zeroes everywhere except in the sign bit. For the DEC-20, the mask `0x800000000` must instead be written as `(1 << 35)`. An indirect function call like `(*(fontptr->charxx))(c,outrow)` results in a jump to the word containing the function address, instead of the function itself. Introduction of a temporary variable for `fontptr->charxx` works around the bug. Right shift of a signed value does not propagate the sign bit. Most C compilers propagate the sign of a signed value, and zero-fill an unsigned value; draft ANSI C leaves the behavior for signed values up to the implementation. Since the DVI file is full of 8-bit, 16-bit, 24-bit, and 32-bit numbers which may be either signed or unsigned, correct reconstruction of a full-word integer of the appropriate type is required a lot, and this involves shifting to propagate the sign. The PCC-20 preprocessor and compiler tables turned out to be too small for parts of the drivers, but since I had source code available, I was able to rebuild the compiler to remedy this.

With KCC-20 on the DEC-20, the drivers uncovered a compiler bug; there

---

optimization, compilation takes about 50% longer, but the output `.exe` files are now 3Kb to 7Kb shorter, and string functions are expanded in-line. I must also report that I found four compiler bugs during the first weekend of using it.

was exactly one instance of an incorrect instruction generated—assignment of a float to an integer was done by a rounding (FIXR), instead of truncating (FIX), instruction. This was a critical instruction, since it was used in the code section that converted magnification values to a text string forming part of the font name. A debugger patch on the executable programs was used until the compiler developers fixed the problem.

The GNU project's gcc compiler, which can generate code for both Motorola and VAX architectures, found one bug in the driver code, and the drivers revealed two bugs in the compiler in the Motorola 68000 code generation! All three were rapidly fixed.

With 4.1BSD Unix, a rather bizarre bug was uncovered which took over a year to finally track down. The drivers would run correctly up to the point of the return from the main program to the run-time library caller (determined by the insertion of print statements), and then would core dump. When run under control of the sdb or adb debuggers, they terminated normally! We finally found that the file close function, which was *explicitly documented* to do nothing if the file already was closed, actually laid a time bomb that blew up after return from the main program. Putting in a test for an already-closed file before calling fclose() defused the bomb.

VAX VMS has a history of having excellent compilers, and benchmarks that we carried out last year for the evaluations of several candidates for a major campus computing upgrade bore out the fact that the Fortran compiler, at least, produces superb code which often cannot be improved by hand-crafted assembly code.

Our experience with the VMS C compiler has been that it is quite reliable, but the library has some atrocious problems. VMS has a record-oriented file system, and with VMS Version 4.0 in 1986, introduced new stream file formats that look somewhat like Unix files. It turns out, though, that many system utilities, like print spoolers, do not correctly support these new formats, so the drivers have been revised to produce only old-style variable-length record text files, or fixed-blocked binary files. We found, however, that the random access routines did not correctly position in binary files. The standard exit() function return codes were VMS-ese, instead of Unix-ese like every other C implementation seems to follow. The ungetc() function for pushing lookahead back into the input stream only worked for 128 out of 256 possible byte values. Some needed functions, like unlink() for file deletion, qsort() for sorting, and system() for command execution in a subshell, were absent from the library. The read() routine will only return 512 bytes at a time, even if you ask for more (every other C implementation I've seen will get as many as you ask for, unless there is no more data in the file).

Perhaps the worst bug (which is apparently documented as a feature in some editions of the manuals, though I have never been able to find it) is the treatment of a simple statement printf("%s",longstring), which simply asks for the string to

be printed. If the string is longer than 256 characters, or some number of about that size, the stack is wiped out, and the VMS last chance error handler takes over and informs you that it has reinitialized the stack, so even the debugger cannot help you. The whole point is that the printf() argument is a 4-byte string *pointer*, and with pointers, the original data should *never* have to be copied to a temporary, which is apparently what the run-time library is doing.

This bug was rather hard to track down, because once the call stack is destroyed, you have no way of telling where it happened. Fortunately, the debugger has an option to trace all function calls, so I started a Telnet log, ran the debugger, and collected a half-megabyte log of call traces. When the stack was finally wiped out, I nailed the offending printf() call, and was subsequently able to reproduce the bug with a one-line C program. Since there is another library routine, fputs(), that prints strings, and it works correctly, I revised the drivers to use it instead for all potentially long strings.

VMS binary files are stored as 512-byte fixed block records. The file system does not directly maintain a file byte count, but it can be reconstructed from the block count, and the offset of the last byte in the last block. Unfortunately, most utilities assume binary files have completely filled blocks, and in fact, the operating system will discard a binary file which is closed with the last block incomplete. Consequently, it has been necessary to assume that ASCII NUL padding has been used at the end of binary files, and the DVI drivers must have special code to ignore this padding in font files and DVI files, and to ensure that before closing a file, enough padding bytes are supplied to fill the last block. Fortunately, in C, the run-time library I/O data structures are accessible to the programmer, so it does not take much code to add this feature. In other languages, it would probably be necessary to interpose a private level of code to buffer output and maintain a byte count, so that at file close, the requisite amount of padding could be supplied.

Most C implementations declare in stdio.h the maximum number of open files permitted by the run-time library, although the names they choose for this limit vary. Unfortunately, VMS has yet another misfeature that makes this number unreliable—quotas, on lots of resources, including the number of open files, and that quota may be less than the limit set by the C run-time library. Version 2.10 of the drivers therefore includes code to determine the limit dynamically, in order to avoid attempting a font file opening when it is known beforehand to be impossible. Previously, a user with a low open file quota might be unable to run the drivers successfully on a document with many fonts, unless perhaps it was done piecemeal to limit the number required at one time. The drivers cache open font files; they just need to know how many they are allowed to have open to do this successfully.

This survey should have given the reader some appreciation for the grief, pain, and suffering we software developers sometimes have to go through. There have probably been more system bugs detected in the DVI driver development

than bugs in the DVI code itself.

Errors and Early History

My first exposure to TEX was a talk that Don Knuth gave at Xerox PARC Laboratories, probably about the summer of 1979. I was in the Bay area for a conference, and since I had read, and greatly admired, the first three volumes of his *The Art of Computer Programming*, but had never met him, I was excited about hearing him speak.

His talk was entitled *The Errors of TEX*, and in it he discussed the software development process based on the careful log he kept of errors in the implementation of TEX, which at the time was in the Sail language, available only on DEC-10's and DEC-20's. Since our local machine is a DEC-20, this let us get TEX at a very early stage, but we had no output devices for a long time, and in my junior position, I hadn't the influence to get the money collected to buy a 200-dot/inch electrostatic printer, or one of the Xerox Dovers.

The growing realization of the enormous importance of the TEX effort led me to become an enthusiastic supporter of the TEX Users Group, and later, when I could find the time, to develop a TEX DVI driver myself for the low-resolution ($60H \times 72V$) Printronix 300 line printer, starting with Mark Senn's driver for the BBN BitGraph terminal (which, incidentally, is about as far from a bit-mapped printing device as you can get). Commercial DVI drivers did not then exist, and even if they had, our extremely tight budget situation would have precluded acquiring them.

By the time I had added support for a couple of other printers, it became clear that maintaining similar, but separate, drivers took a prohibitively large effort, and the concept of a driver family was born. My background in computational theoretical chemistry has made me extremely sensitive, and well read, about the subject of software portability, and I confess, with some pride, to holding the general attitude "If you cannot program it portably, don't bother to program at all!". Until the late 1970's, the only language in which one could do remotely portable programming was Fortran, in which I consider myself an expert. For the reasons described above in the discussion of why I chose C, and not Web, it became evident that the only other candidate for writing portable software was C, and the growing popularity of Unix, and workstations, large minicomputers, and supercomputers using Unix, seems to indicate that this trend will not soon be reversed.

I too have kept a log of changes made to the drivers, although only since the summer of 1986 when I first began to make them available to others. Most changes noted in this revision history (stored as the file 00REVHST.TXT in the distribution) are for addition of new features, or workarounds for compiler or

operating system bugs or misfeatures. There are a few programming errors which have shown up and been corrected; here is a list of most of them.

- The code assumed that the TEX \count register values stored in the DVI file were *unsigned* 32-bit values; they are actually *signed*.
- Support for the page number range step option was added, but I forgot to take out the original code which reversed them if the initial value was larger than the final value.
- I added an optimization in PostScript output to remember the size of the last rule set, so that a following rule of the same size could be set with fewer commands (this is quite valuable for the LaTEX Bézier picture mode option, which draws curves by typesetting many small rules), but forgot to reset a flag when a new DVI file was started to cause the rule size to be output again.
- The command line option parsing routine called an error routine to write its messages, but that error routine tried to open an error message file whose name could only be determined after the command line was successfully parsed.
- The hand-optimization of collapsing of two successive loops of the same range into one resulted in an array getting indexed by two's, instead of by one's.
- Ports to the Sun and VAX VMS environments caught some instances of dereferencing null pointers or incrementing them, and of passing null string pointers where null strings should have been used. A lot of even relatively mature C code has been bitten by ports to these two machines. Most older C libraries treated a null string pointer argument as equivalent to a null string; the Sun library core dumps instead.
- The DEC-20 KCC compiler was the first C compiler I had used for which character pointers were not represented like integer pointers, and the run-time type checking which the KCC developers thoughtfully provided in the library caught some instances of missing type casts.
- A few off-by-one errors have surfaced on occasion. The most serious was an array size which was one too small. It was an array allocated on the call stack, and since stacks on the DEC-20 grow upwards, writing beyond the end of such an array only overwrites unused space. On the Intel, Motorola, and VAX architectures, stacks grow downward, so writing beyond the end of a stack array is likely to wipe out the calling history, causing the program to crash when the current function attempts to return to its caller. This overwriting could only happen for a TEX document with wide pages, which is why the bug existed for over two years.
- The character painting raster operation on the BBN BitGraph driver was set so that it worked correctly with black characters on white, but not the reverse (which I don't like, and consequently, never used); black characters on black are rather hard to read!

- C's case sensitivity let one bug survive for a time—there is a preprocessor symbol named SUBPATH and a variable named subpath, and there was one instance where the wrong one was used.
- The driver code keeps track of the number of open font files, and increments a counter before the file open attempt. Unfortunately, I forgot to decrement the counter again if the open failed. Since font file open failure is uncommon, this bug existed for quite some time.
- A bug was caught in September, 1987, which affected only one driver (out of twenty) in the family, and only with one of the three supported font file formats, the GF format; some character metrics were referenced before they had been read from the font file. I never found this locally, because we normally use the more compact PK format, and earlier this year, removed all font files in the old disk-hogging PXL format.
- When I added support for font files containing up to 256 characters (Computer Modern fonts use only 128), to handle European and Japanese extended character sets, I thought that all that was needed was to change a couple of macro definitions so that 127 and 128 became 255 and 256. Unfortunately, there were several loops whose indexes ran from 0 to 255, but the loop index was declared of type BYTE (a private type corresponding to unsigned char); the last loop iteration incremented the index from 255 to 256, which of course is equivalent to 0 as an 8-bit byte value, so the loop became infinite. The bug was not noticed on my main development machine, the DEC-20, because all integers are stored as 36-bit values.

Performance

A considerable amount of care has gone into the preparation and programming of the DVI drivers to ensure adequate run-time performance. The consistent use of the smallest possible integer data types has already been cited in connection with the Intel iAPX architecture used in the IBM PC family. Run-time profiling on the DEC-20 and Sun systems has been used to identify hot spots in the code, and I am reasonably comfortable with the code efficiency.

The VIRTUAL_FONTS code added in late 1986 provides for the reading of entire font files with a single read request. The buffer size is dynamically matched to the exact file size. Subsequent random-access positioning through calls to fseek() then only result in manipulation of pointers into the buffer, rather than input requests. This is important when font files reside somewhere else on a distributed system network, because there is a relatively large latency involved in I/O transfers. Particularly in the PK font file format, font file access involves many seeks to relatively small data packets.

A similar optimization for the input DVI file and output files has yet to be

made. The machine that shows the greatest improvement in disk transfer rates with increasing data size is the IBM PC, where a speedup factor as large as 9 can be obtained by increasing the block size from the default of 128 bytes to 8K bytes; regrettably, this is also the machine with the least amount of memory available for this optimization. Even so, the effect of such an optimization would only be a few percent, since the driver profiles show that less than 15% of the total time is spent in reading and writing files.

DVI driver execution times depend heavily on the number of fonts used, and on the amount of output produced. On a Sun 3/280 system using the TEXbook as a test case, the DVI drivers for laser printers run about 3 to 4 times faster than TEX itself.

Commercial, or Public-Domain?

This section is an editorial on the question of proprietary and commercial software, versus public-domain and free software.

Some users have raised the questions "Aren't you taking business away from those firms who have chosen to provide TEX and TEXware commercially?", and "Isn't it unfair to take software developed at a university and effectively compete with commercial software?" To both of these, I must answer a qualified yes. Indeed, there are certainly a few users who got these drivers at little or no cost, and who otherwise might have purchased a commercial package.

I defend my offering, however, with two important points. First, nobody has to pay for TEX or LaTEX from one of the TEX site coordinators, apart from a modest media distribution charge; those with network access never even have to come up with hard money. Yet there are several companies, small at present, but growing, who find that they can indeed make an adequate living selling TEX implementations and TEX support.

On the question of university research efforts competing with commercial products, I suggest that first, university researchers have an obligation to the tax-paying public who support them to make the results of their work freely available to others, and that second, in science and technology, this is often how commercial firms get started.

Much of the computing and software industry began at universities. So did Emacs and TEX, and I believe that neither of them could ever have been developed in any commercial environment, where projected market returns and the methodology of design-specification-before-programming would never have let them be born. Emacs and TEX evolved and flourished through the free exchange of ideas and software by a great many individuals; I do not think any human could have designed either in complete isolation. Nor do I believe that they would have the widespread respect and influence they enjoy, had they been

commercial products.

At this conference, two of the speakers from the commercial world stated that one of their reasons for choosing TEX over a proprietary typesetting system was precisely the fact that it was not tied to any particular company, upon whom they would then be economically dependent, and that it offered them the choice of a variety of public-domain and commercial software suppliers.

My second point is that I will not be here forever; apart from questions of longevity, I may choose at any time to go off and do something else. I do hope that when that time comes, the DVI driver family will have become mature enough to survive *as a family*, and not have degenerated into distantly related cousins who hardly acknowledge their common ancestry. During the past year and a half, a very large portion of my time has been devoted to this work, and since my naive public announcement of their availability in January, 1987 (I expected perhaps a score of responses, not a score^2!), the burden of filling orders and answering telephone, postal, and electronic communication has become almost onerous. I can do it because my position at Utah carries a considerable amount of freedom to do what I think is useful and interesting, and there have been ample local benefits to justify the effort, even if I had not decided to distribute the family more widely. I continue to hope that the alternate distribution channels noted in the next section will remove much of the load.

I confess that I am still unhappy with the high cost of personal computer implementations of TEX, but that is a general problem in the industry, and I do not fault commercial TEX suppliers for doing what everyone else is doing. For example, where before we could supply access to a Fortran compiler to users at a very low average software cost (less than US$2 per user per year on our DEC-20), we now find that a personal computer Fortran compiler costs several hundred dollars per user. Even if we buy one copy for each half-dozen users, who may not then use it simultaneously, we still are paying very much more per user than we do for access to the mainframe compiler (which also is of considerably higher quality and reliability). But we are living in a new age—the information age—where information and knowledge now carry a price tag. Except for software, I cannot think of any other commodity in mankind's history that, once developed, has had essentially zero cost† of reproduction. This situation is completely new, and rules and customs for dealing with it will take society time to work out.

An end user who is competent on a local computing system should have no great difficulty in installing the drivers, and one with C programming experience will even be able to add support for new devices. Many TEX users are not, however, in this class. Individuals without much computer experience may need

† Music and video recording are close examples, but so far, their mostly analog form has generally precluded distribution over networks. For digitally-recorded music, the question of unauthorized reproduction has led to public debate on the desirability of legislative regulation of the sales of digital recording devices.

more support, and a commercial supplier may be more satisfactory. Large organizations who feel the need to make a group commitment to a software vendor may also feel more comfortable with that approach.

I am probably conceited enough to believe that my code is of high quality, but as one individual, I cannot possibly provide the support that a professional staff at a company can.

Another issue that users may wish to consider is extra features, or "bells and whistles". Some vendors have gone to considerable effort to integrate their software smoothly into a particular machine environment (such as those that make use of the Xerox desk-top model and windows), or to perform all kinds of neat optimizations internally to make the software run faster. For many people, those features are worth paying extra money for.

Commercial firms generally have to recover costs, and a user with a brand X printer on a brand Y personal computer may be told that there is just not the market to justify supporting it. This has already happened to some DVI driver users that have spoken to me, and I think it is appropriate that public-domain software is available to fill their needs.

Availability

The DVI driver family can be obtained from a number of sources. The preferred method is via electronic network, since that does not involve human effort. This does require some sort of file transfer system, such as ARPANet FTP; the distribution is much too big to send via electronic mail. Failing that, the TEX Users Group site coordinator for your machine class may be able to provide it as part of the normal TEX distribution. Here is a list of the current distribution channels:

ARPANet:	ANONYMOUS FTP to SCIENCE.UTAH.EDU (TOPS-20) or CTRSCI.UTAH.EDU (VAX VMS);
Australian ACSnet:	Contact munnari!latvax8.lat.oz.au!ccmk;
British Janet server:	Contact AbbottP@uk.ac.aston.mail;
European Bitnet server:	Contact rz92@dhdurz1.bitnet;
European DECnet server:	Contact Calvani@vaxfpd.infnet (reachable through cernvax.bitnet).
IBM PC floppies:	Personal TEX, Inc;

Japanese JUNET and North American Bitnet redistribution channels should soon be available.

For ANONYMOUS FTP (password GUEST) to either of the Utah machines, get the file 00README.TXT in the login directory; it contains details of how to retrieve the DVI distribution, and other TEXware and public domain software.

For the convenience of Unix sites, a compressed tar file is kept in the master archives on SCIENCE.UTAH.EDU. VMS BACKUP savesets are kept on CTRSCI.UTAH.EDU.

To permit easy updating of source files, each directory contains files with alphabetic and reverse time-ordered directory listings, and corresponding FTP command files to retrieve the files in either of those orders.

There is also a tape distribution service handled directly from my office in Utah. The available formats are:

- ANSI D-format;
- TOPS-10 BACKUP;
- TOPS-20 DUMPER;
- Unix tar;
- VAX VMS BACKUP (1600 bpi or 6250bpi).

Except as noted, the tape density is 1600bpi. ANSI D-format is not recommended, because it does not provide for directory names on the tape, only file names; extraction must be done carefully, since there are files in different directories which have the same name. The other formats preserve directory structure, file names, and file write dates. Send a 2400ft 9-track tape with a cover letter indicating the preferred format and density. The tape distribution normally includes a large set of Computer Modern fonts computed with \mode=imagen, since it has been our experience that many sites do not yet have METAFONT running to produce their own.

Redistribution sites may levy a modest fee to cover their costs. At Utah, we appreciate donations; this is all a volunteer effort.

I maintain an electronic mailing list and as of November 1987, 14 issues of a newsletter (in 8 months) have gone out to about 170 subscribers. If you have, or acquire, the distribution, you should find it useful. Send your request to BEEBE@SCIENCE.UTAH.EDU. Back issues are present in the distribution as files 00MAIL.*. I regret that manpower and budget limitations do not permit postal mailing of printed copies to other sites.

I do not feel that it is appropriate to place technical discussions of this driver family in other TEX-related bulletin boards, such as TEXHaX or INFO-TEXT; however, short announcements of major developments may be posted in those forums from time to time. If you desire, comments sent to me can be posted in the next newsletter.

At the time of writing, we have distributed about 250 copies of the driver by post from Utah, and based on file access counts (which TOPS-20 so considerately maintains), perhaps about 150 copies have been retrieved via ANONYMOUS FTP. With some end-user redistribution, and redistribution from the sites listed above, it seems realistic to assume 500 sites have obtained these drivers.

Additional Support Software

The DVI driver distribution contains a small amount of additional support software, all of which is in the public domain:

make	Unix-like make for six different operating systems;
lptops	line printer to PostScript converter;
lw78	PostScript spooler driver;
hd and unhd	dump and undump utilities;
errshow	merge source and errors from Microsoft C;
keybrd	keyboard input package;
vaxvms	VAX VMS C flaw workarounds;
wlstex	WordStar to LaTEX translator.

The make utility is an important addition, since it allows automatic building of any of the drivers on each of the supported operating systems.

Ordinary hexadecimal or octal dump programs for output device binary files are less useful than hd, which produces line breaks for each escape sequence, and permits embedded comments. unhd can turn the output of hd back into a binary file, which makes editing a binary file quite convenient. When things go wrong in the device output, it can be a very tedious exercise to mentally decode a dump of the file, and having it in a form which can be edited, and in which each device command starts a new line, makes this task much easier.

Credits

This article would not be complete without proper credits to people and institutions who have made contributions to the family.

Mark Senn (Purdue University) wrote the original dvibit in C, from which it all began; he based his work in turn on David Fuchs' (Stanford University) definitive dvitype program. Stephan v. Bechtolsheim and Bob Brown (Purdue), Robert Wells (BBN), and Jim Schaad and Richard Furuta (University of Washington), improved dvibit before I began on it. Simon Barnes (Schlumberger Cambridge Research Ltd.), and Robin Rohlicek (BBN) provided useful additions to dvibit which were generalized and incorporated in Version 2.07.

Contributions for PostScript devices came from Neal Holtz (Carleton University, and Barry Smith (Kellerman & Smith) helped out at an early stage with a correct description of the font downloading mechanism (which was incorrect in Apple's documentation).

John Sauter (DEC) prepared dvil3p and dvil75, using one of the existing drivers as a model.

Lon Willett (University of Utah) wrote dviimp, starting from the Hewlett-Packard LaserJet Plus driver, dvijep.

Marcus Moehrman (Universität Dortmund) contributed dvie72 and dvieps.

Matthias Moritz (Katholieke Universiteit Nijmegen) implemented the driver family on the Atari 520ST+.

I would also like to thank the sites who have offered resources to support electronic redistribution of the DVI driver family.

If I have missed anyone, please accept my apologies.

Many others world-wide are helping to keep my electronic mail volume at 840Kb/month in 1987! Thanks to them all for their input.

Bibliography

ADOB85 Adobe Systems, Inc., *PostScript Language Reference Manual*, and *PostScript Language Tutorial and Cookbook*, Addison-Wesley (1985).

ANSI66 *American National Standard Programming Language FORTRAN*, ANSI X3.10-1966, American National Standards Institute (1966).

ANSI78 *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978, American National Standards Institute (1978).

ANSI83a *Military Standard Ada Programming Language*, ANSI/MIL-STD-1815A-1983†, American National Standards Institute (1983).

ANSI83b *American National Standard Computer Programming Language Pascal*, ANSI/IEEE 770X3.97-1983 (also British Standard BS 6192-1982, and ISO 7185-1983) (1983).

ANSI86 *Draft American National Standard for Information Systems—Programming Language C*, ANSI X3.9-1978, American National Standards Institute (1986).

ANSI87 *Draft American National Standard Programming Language Fortran 8x*, ANSI X3.9-1978, American National Standards Institute (1987).

BENT86 J. Bentley, "Programming Pearls", Comm. ACM **29**, 471-483 (1986).

DACR87 F. da Cruz, *Kermit—A File Transfer Protocol*, Digital Press (1987).

HARB87 S. P. Harbison and G. L. Steele Jr, *C: A Reference Manual*, 2nd ed., Prentice-Hall (1987).

† 1815 is the year of the birth of the world's first computer programmer, Ada Augusta, Countess of Lovelace, colleague of Charles Babbage, and daughter of Lord Byron.

JENS74 K. Jensen and N. Wirth, *Pascal User Manual and Report*, 2nd ed., Springer-Verlag (1974); revised by A. B. Mickel and J. F. Miner, 3rd ed., Springer-Verlag (1985).

KERN78 B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall (1978).

KNUT84 D. E. Knuth, "Literate Programming", The Computer Journal **27**, 97-111 (1984).

KNUT86 D. E. Knuth, *Computers and Typesetting*, *Volume A: The TEXbook*, *Volume B: TEX: The Program*, *Volume C: The* METAFONT*book*, *Volume D:* METAFONT*: The Program*, *Volume E: Computer Modern Typefaces*, Addison-Wesley (1986).

LAPI87 J. E. Lapin, *Portable C and Unix System Programming*, Prentice-Hall (1987).

ROCH85 M. J. Rochkind, *Advanced Unix Programming*, Prentice-Hall (1985).

STEE84 G. L. Steele Jr., *Common LISP—The Language*, Digital Press (1984).

WIRT76 N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall (1976).

WIRT83 N. Wirth, *Programming in Modula-2*, 2nd ed., Springer-Verlag (1983).

The Use of TEX in a Commercial Environment

DAVID NESS

TV Guide
Radnor PA 19088

ABSTRACT

Using TEX in a commercial environment involves both *pain* and *pleasure*. This paper describes some of both. It concludes that while the *pleasure* outweighs the *pain* as is, the *pain* could still be reduced significantly.

Purpose

We have been using TEX in a 'commercial' environment for nearly three years. We have also been experimenting with METAFONT for a little less than one year. The purpose of this note is to record some of both the rewards and frustrations that we have encountered during this experiment.

What Led to the Experiment

At TV GUIDE we render a lot of type. We print in the neighborhood of *one billion* copies of our magazine each year. During the year we render about *one million* different pages, and bind them into *five thousand* different 200 page magazines.

The job of producing this output requires a complex coordination of both human and automated processes. It is—needless to say—not a simple matter to obtain information from nearly *two thousand* TV stations nationwide and get it laid out and printed in a timely fashion in a magazine which is delivered to nearly *twenty million* American homes.

Experimentation with Type

In the course of experimenting with the look and readability of this magazine, we have many occasions to manipulate type. This involves setting and proofing lots and lots of different formats and styles of type. As those who have done

this extensively know all too well, this can become quite expensive. Doubly so when we have to train people in particular typesetting technology only to find that either the people leave or that the typesetting technology becomes obsolete and replaced by newer and better stuff—which then requires a retraining of the staff.

Experimentation with type takes many dimensions. It is particularly useful to be able to develop page formats that test hypotheses about the legibility and readability of different page layouts and structures. Even seemingly subtle changes can have enormous consequence when you are printing a *billion* copies. In our situation saving an average of one line every 10 pages saves 300,000,000 pages/year.

Adapting to an Unknown Future

It is also clear that typesetting technology is in a great state of flux. Within the last *five years* the cost of the equipment required to set type of the quality presented in this memo has dropped from more than $100,000 to well under $5,000. At the present time there is little sign that this dramatic rate of progress is slowing down. Both scanning and printing equipment seems to be improving at a great rate. Every day we seem to be able to do more, faster and cheaper.

This makes it difficult, if not impossible, to make much of a guess where things will be a few years from now. Major players in the typesetting business could well fall by the wayside as new companies come to the fore. Three years ago the 'Canon Engine' was in a preliminary announcement from a "copier company." Now Canon is a major player in the printing business, and have delivered hundreds of thousands of their engines under Canon, Apple, Corona, QMS, *etc.* trademarks.

TEX affords us reasonable insulation from this innovation by virtually guaranteeing that we will be able to participate in the best of it. When we started our TEX project Allied wanted in excess of $10,000 for a screen preview device. Now we get higher quality from an $800 Wyse terminal (about 1000×1000 pixels) than we would have gotten from that older device. Today 300 *dpi* laser printers cost us about $1,200.

Since the TEX community is so broad—and so competitive—innovation is (1) passed along quickly; and (2) major benefits of innovation are passed on to the consumers, rather than taken in profit improvements by the manufacturers.

Public Domain—Second Sources

One of the reasons that we decided to use TEX for our experimental environment was that the code for TEX is in the public domain. This meant that—worse comes to worse—we could always track down and innovate our own way around any particular problems which happened to arise.

Further, wide public availability of TEX has produced both considerable innovation and alternative sources of supply in the TEX world. In the PC versions of TEX we are offered two widely available products, both of which appear to be

very faithful to the letter and the spirit of the original version. We have made use of this, particularly in terms of product innovation, in many directions which were unanticipated at the time that we made our earliest TEX decisions.

The publicly available documentation, *The TEXbook*, The Joy of TEX, LATEX, *The PC-TEX Manual*, *Another Look at TEX* and—of course—issues of *TUGBoat* provide lots of valuable information, examples and clues on how to use these powerful facilities. This is much more help than is available for most of the other typesetting systems. *Beginner's Guide to TEX* and the other helpful (and sometimes operating system specific) documents published by schools and manufacturers also provide a good secondary source of information.

Finally, there is the sharing of information accomplished through TEX courses, local user support groups and—last, not least—TUG itself that help us survive some of the frustrations and equally (or more often) help us share the 'fruits' of TEX labors.

Our Fonts

TV GUIDE requires some unique fonts. For example, we print the numbers of the TV channels with a little 'TV screen-like' faces. Obviously we have these characters in our fonts on our traditional typesetting equipment. However, we are not very likely to find them on any proof equipment, particularly those with built-in fonts. While we could, no doubt, induce some interest on the part of some of the suppliers with a sufficiently large payment, it is much nicer to have the support of the broader TEX community to help us produce these characters. We have been able, for example, able to develop some general-purpose software to help us create some of the characters by hand. This allowed us to produce some fonts in standard resolutions at standard sizes. Now the arrival of METAFONT affords us the opportunity to develop our fonts with resolution and scale independence.

However, using METAFONT isn't easy, and it takes a substantial investment of time an energy before one can gain any but the simplest control over the METAFONT output. This piece of software has an unusual characteristic. One gets something important (all of the CM fonts at all reasonable scales and resolutions) for just about nothing. However, then to go beyond this seems to require a **very large** investment before further fruits are obtained.

Producing Listings and Grids

We experiment a lot with different alternatives for listings and grids. There are a large number of variables associated with getting a reasonable display of type, and it sometimes takes many attempts before satisfactory results are obtained. TEX allows listings and grids to be produced very efficiently, at a low cost. More importantly, it reduces the turn-around time for producing type from what was often days to a few minutes.

Particular and Peculiar Problems

One of our toughest problems deals with the width of a column. As Knuth's beautifully typeset books show, good algorithms allow one to almost completely avoid hyphenation and ugly space expansion or shrinkage when setting material at 29 picas. However, when you are setting columns at less than 12 picas, then some ugly hyphenation or spacing is inevitable unless all of the text is actually re-written to fit the space. If we are not careful we regularly create text which is 'intolerably ugly' by TEX's normal standards, and learn to become more and more tolerant of the intolerable.

Layout

One of our major problems with TEX has to due with the difficulty of dealing with layout problems. This is not to say, of course, that TEX can't handle these problems, but rather only to observe that there are not any very direct mechanisms in TEX or any of the (so far released) macro packages ($\mathcal{AMS}$-TEX or LATEX) to deal with layout problems.

There seem to be some intellectually interesting problems associated with layout, and I remain hopeful that some members of the TEX community will find themselves attracted to the problem. Since TEX has such substantial capability for typesetting, it is a shame not to be able to use it easily in the layout context where it might be so helpful.

Re-Encountering Text

We occasionally also need some relatively straightforward way to re-encounter some text after making some placement decisions. For example, in magazine typesetting it is not unusual to have pages that are set two-up followed by pages which are set three-up. When TEX encounters such a configuration, it has a tendency to complete the setting of a full paragraph at the width appropriate for two-up pages before being willing to adjust its `\hsize` to the width appropriate for three-up.

It is clear that imposing badness penalties across a break of this kind is conceptually quite a difficult matter. It would be difficult coding, indeed, to define—much less implement—a notion of 'best' setting across such a two/three split. However, in general we are only interested in 'good' solutions to this problem; we need not worry if the 'best' eludes us. In most cases we would be quite happy to let TEX decide on the paragraph break with its conventional methods, if that could then be followed by re-setting the type after the break at the new width.

We have found it difficult to grapple with this problem, although a letter from Knuth contains some promising suggestions. We are trying to understand the full implications of his suggestions for this problem, and it isn't easy. However, with help we seem to be making some progress.

TEX, Programs and Documentation

The relationship between TEX and (computer) programming activities first occurred when we began to use TEX to prepare our program documentation. It proved to be very helpful when we could use typewriter-like fonts in the documents to highlight input and output displays. Once TEX became a normal part of the working environment it was natural to extend its capabilities to helping us with the actual display of programs themselves. We developed the `CTEX` and `INDEX` programs that produce very nice listings of our source code. This has allowed us to change our standards for code documentation without requiring a real 'revolution' in the way that we write code—something that would be required if we were to convert to `CWEB`, or any similar environment.

We also have incorporated the documentation associated with the programs into a more general documentation system. This system allows us to keep track of documents, their version, their status, *etc.* One of the nice things about TEX is that it is easy to incorporate as a part of a process which finds the document source, unarchives it, typesets it and then renders it on a typesetter or laser printer.

We also make use of the fact that TEX allows us to easily describe and set documents that are independent of their final physical size. Sometimes $8\frac{1}{2}in \times 11in$ documents are convenient. However, often the new computer standard of $5\frac{1}{2}in \times 8\frac{1}{2}in$ proves to be desirable. TEX conveniently allows us to largely separate the task of creating and maintaining the document from the tasks associated with rendering it.

We are also running an interesting experiment in having users read code. This is part of a general attempt to get our programmers to indicate their **intent** in their program documentation. Since the majority of our users are editorial professionals, we get lots of helpful (and painful) criticism of the English that we use in documenting things.

De-Tuning TEX to Match Conventional Typesetters

We also have an interesting situation where the high quality of type set by TEX gets in our way. In some circumstances we want to set the equivalent of galleys that are to be cut to specific lengths later in the composition process. Conventional typesetting technology allows this in a rather simple way. We typeset the longest text and then simply observe whether shorter renderings of the text are legal (*i.e.*, they don't produce widows, for example) by pretending to 'cut' the text to shorter fragments. When we go to final typesetting, we are able to tell the typesetter to set type to the appropriate length, and we know exactly what we get.

If we cut a TEX paragraph, however, TEX will re-set (perhaps all of) the lines of the text in a different fashion. TEX may well decide to move some words up or down from line to line to get a better looking paragraph now that the text is known to be shorter. When we want to mimic conventional (forward only) typesetters this is not desirable.

What we need to be able to handle this is an ability to de-tune the TEX algorithms so that it doesn't consider flowing the text back up into the paragraph after the cutting decisions are made.

Democratization of Typesetting—Decentralization Issues

TEX allows us to both 'democratize' and 'distribute' typesetting tasks. This represents a substantial revolution in the process of getting ink committed to paper.

First, TEX allows us to put typesetting capabilities into the hands of a wide community of possible users. These users can range in sophistication from people who know little or nothing of typesetting to old-line typesetters who understand the business quite well. There are many levels of TEX which correspond to this. One can know no more than to just put 'funny looking' things (with backslashes and the like) into the text, and still find TEX quite useful.

Second, and perhaps even more important, TEX allows us to distribute the typesetting capability down to individual users. We are economically able to put the ability to look at type into the hands of virtually every user who has a PC at their disposal. This allows us to distribute typesetting capability to each of our remote offices, printing sites and anywhere else that it might prove to be useful.

The Separation of Form and Content

If there is a big point to be made by adopting the TEX environment, it is this: *Separation of form from content.* Very unfortunately today's trends in typesetting are running precisely contrary to this. There is a naïve pull toward WYSIWYG typesetting that completely *integrates* the content into the form. For our purposes—and indeed, I think most other purposes as well—this is clearly an error. We very definitely want the *content* of what we do to be quite independent of the *form* in which it will finally be rendered. Only in this way can we defer decisions about commitment to ink and actual type as long as is reasonably possible in the publishing process.

One can only hope that it will not take too terribly long to realize that—for the most part—we should provide ourselves with an environment that allows us to separate worries about the form from worries about the content. TEX makes this not only possible—it makes it quite convenient.

The Arguments Against TEX

In our environment TEX encounters a lot of resistance. As with many other great ideas this is no surprise. Let's treat the comments and their likely background and cause one-by-one.

TEX is for mathematics. I guess TEX *is* for mathematics. The other day I set six pages of beautiful notes from a calculus class that I took in my undergraduate days. However, it is simply poor reasoning to think that because TEX *is* for mathematics, it *isn't* for something else. It works pretty well in lots of different environments.

TEX is for books. TEX also *is* for books. I guess it is a fair observation to suggest that Knuth has paid more attention, at least in print, to considerations relevant to producing books than for other varieties of typesetting. Perhaps if he had been principally interested in typesetting articles for the popular magazines, we might have an emphasis on other facilities in TEX. However, it is my feeling that the complexity of many of the problems faced in book production assure us that comfortable solutions associated with magazine production are likely to evolve.

TEX is too complicated. TEX *is* too complicated. But again, putting ink on paper is an extremely complicated process if one wants to be able to control it both exactly and with considerable generality. Each time that I learn more about how to use TEX in some complicated situation I become more convinced that the complications are necessary, and indeed desirable. The alternative would be to give up the control that is actually so important.

TEX is too cheap. It may seem, at first, that this is an absurd objection. Of course, it is never stated in these terms. Yet, quite often it is the case that people with a background in typesetting simply cannot believe that it is possible to set such good type so cheaply. I am reminded of the fact that in the 1940s *Toni Home Permanents* didn't sell when they were first test marketed at 25¢. When the price was raised to $2, they sold like hot-cakes. People who have deep commitments to old-time typesetting technology don't want to believe that it is possible to set acceptable type, with great reliability, on 300 *dpi* laser printers generating your own fonts for a combined hardware/software cost of less than $5,000.

TEX requires specialists. Again, TEX requires *some* specialists, but actually it is less demanding on our environment than many of the conventional typesetting machines. What TEX allows, in direct contrast to more conventional typesetting technology, is for different levels of understanding to be effective for different users.

Thus, these first few objections to TEX do not seem to be very substantial in our environment. We continue to encounter these arguments, however, and it will probably be a very long time until they are no longer brought forward. However, there are a few additional arguments that should be at least discussed.

TEX source is too ugly. This does seem to me to be a valid criticism of TEX. For people who so obviously care so deeply about the appearance and quality of all output, it strikes me as odd to have such an *ugly* input language. While TEX is obviously quite logical in its structure, it is often difficult to make input read easily or look nice. It is mindful of APL or FORTH. Spaces count some places,

and don't count other places, so one has to be a bit careful about breaking lines in macro definitions, or about spacing out definitions so that they look readable.

TEX is too hard to debug. Again, this strikes me as largely a valid criticism. Exactly which space in a macro causes a particular (unwanted) space to appear in some output can occupy hours of tracing down. Further, when a macro definition blows up, a character gets misplaced in an alignment or, (horror of horrors) we overflow our font tables or macro definition space, it can take a considerable amount of work to (1) understand what is going on; and (2) to fix it. Here, again, I have the basic impression not that this is all wrong, but rather just that it is a good deal more complicated than it really needs to be.

TEX is a hard language to program in. This, too, seems to be valid. One need only look at the complexity of the definitions in LaTeX, and at the fact that LaTeX requires nearly $\frac{1}{4}$ million characters of text, to understand that 'programming' in TeX is hard work. Of all of the tasks that need to be performed, detecting and handling error conditions seems to be one of the most difficult. Programming TeX to do things is generally difficult, but nevertheless rewarding.

Of course, some of the macro packages deal with this problem. However, I find that I rarely want all of the constraints of the non-`plain` macro packages. I would sometimes be very happy to be able to pull out the list processing capability or the token-processing parts of LaTeX, for example. The fact that TeX is hard to program in means that it is difficult to find and 'lift' parts of these macro packages without absorbing them *in toto.* This actually makes the point about the 'look' of TeX.

Finally, *TEX needs some 'features'.* We are about to fall into some difficulties along at least two fronts. First, and to us the most important of these areas, is that of *layout.* We somehow need to be able to exact more flexible control over the layout of documents than the very long `\parshape` commands will easily allow. We need to be able to mark flows of text, and then re-encounter the text.

We also need to develop either some 'features' or some philosophy which will help us properly define the role of `.DVI` *vis-a-vis* `POSTSCRIPT`, and how all of this should be related to 'specials' in TeX. If we don't act reasonably quickly in this dimension, we may lose some of the considerable value gained by having had such an early and clear definition of problems and solutions by Knuth.

Where are we going?

We will continue to use TeX in our experimentation. Indeed, we are actually integrating some use of it into operational systems as a regular part of the process of producing some of our type. We are increasing the proportion of the type that we set *via* TeX, and are gradually obsoleting other technologies so that we can focus our support facilities on this particular way of producing type.

Needs

The TeX community is a robust one, but—at least so far—it has not proven to be particularly responsive to the needs of those of us who daily operate in

a commercial environment where deadlines are the rule of the day. The TEX community is *academic*—in both the best and worst senses of that word. TEX help is truly *priceless*, often given freely and quite wonderfully, but sometimes not available at all. Academic concern, quite reasonably, is more often guided by *interest* than by the thought of *reward*. This sometimes means that we can get quite difficult problems solved for free if they are interesting, but often can't get simple problems solved for any price if they are boring.

As a simple example, information that circulates quite freely in the academic community is not necessarily available to commercial users. A lot of TEX related information is shared in formal and informal networks that most commercial organizations are not a part of. Thus we are outside of the mainstream.

Commercial organizations often need to be able to 'count on' things a little more. That's the bad part of using TEX in a commercial environment. Generally, we can also afford to pay for them. That's the other side of the coin.

Literate Programming in C

SILVIO LEVY

Mathematics Department
Princeton University
Princeton, NJ, 08544

ABSTRACT

This paper reviews the idea of literate programming and the basics of Knuth's **WEB** documentation system, then discusses in general terms the use of (a version of) **WEB** with C programming. It also summarizes the author's experience in adapting **WEB** to C; this may be useful to those who would like to try literate programming in other languages. The author's version of C **WEB** is available for distribution.

What Is Literate Programming?

"Literate programming" is a phrase introduced by Don Knuth [1984] to designate a style of writing programs in which documentation and code are of equal importance, and are written more or less simultaneously. One aims not only at writing code that works, but also at explaining to oneself and other possible readers of the program how and why the code works. This is best done by presenting concepts and algorithms in human language and in the order that seems most convenient for exposition to humans—which is often not the order that is most convenient for the use of the computer.

The inclusion of "oneself" in the previous paragraph may sound silly, but is actually crucial: a programmer who goes back to a program he or she wrote six months before and did not document adequately generally finds it as incomprehensible as if it had been written by someone else. Explaining the code to oneself also helps one make fewer mistakes, thus reducing debugging time.

Knuth developed the idea of literate programming several steps beyond the

documentation facilities offered by most programming languages, which generally consist merely of some way of embedding comments in the code. He invented **WEB**, a documentation system consisting of two parts: **tangle**, which takes a source program containing pieces of code and explanatory TEX text and rearranges the code according to the user's instructions, producing a computer program ready to be compiled; and **weave**, which from the same source produces a TEX document, including various useful indexes and fully formatted code.

Thus "**WEB**" is not just a program but a class of programs, like "compiler", in the sense that can be realized in different ways, and for different programming languages [Knuth 1984]. Knuth's implementation was designed for Pascal, and in fact incorporates additional facilities, like macros and string handling, that enhance Pascal's capabilities. The use of this "first **WEB** system" is exemplified in [Knuth 1984] (a fascinating paper, which deserves to be read independently of the merits of **WEB**), and fully described in a manual [Knuth 1983]. The manual is short, because a programmer familiar with Pascal and having at least a nodding acquaintance with TEX really doesn't have much else to learn in order to start using **WEB**: the possibly hard part is to acquire the discipline and the verbal skills necessary to express oneself clearly ...

C Versions of WEB

Pascal presents many shortcomings when used for system programming, not all of which are addressed by **WEB** (see the article by Nelson Beebe in these proceedings). For those of us who would like to write literate programs in other languages, the solution is to adapt **WEB**.

In the spring of 1986 I set out to write such an adaptation to C, because the only existing one at the time (to my knowledge) used **troff** as a typesetting language [Thimbleby 1983], which seemed a bit of a step backwards. I worked on it part-time for about four months until I had something that satisfied me. Since then at least one more version of C **WEB** has come out [Guntermann and Schrod 1986]. For technical details of my version the reader is referred to [Levy 1987]; here I will limit myself to discussing a few points of wider interest and illustrating the use of C **WEB** with real-life examples. (These examples illustrate features inherited from Knuth's **WEB**, unless otherwise stated.)

C has a well-developed macro facility, handles strings reasonably well, and allows conditional compilation. There go three reasons to choose Pascal **WEB** over standard Pascal. Is there a reason, then, to choose C **WEB** over C? The answer, for me, has been an emphatic "yes". Here, for example, is the **WEB** description of a routine from the program I've been working on most recently:

---

7. The next function reads from the standard input the data for a triangulated polyhedron and builds the structure representing it. Various types of data are read, each according to a conventional format described below; roughly speaking,

we start with the combinatiorial information (number of triangles and edge identifications), then move on to the geometric information (total angle at each vertex and enough vertex positions to uniquely determine the metric).

```
⟨ Function definitions 7 ⟩ ≡
 read_poly( )
 {
   ⟨ Read n_faces and initialize pointers in the first n_faces elements of f 8 ⟩;
   ⟨ Read edge information and fill the rest of the face structures 9 ⟩;
   ⟨ Go over faces, concatenating vertices in cycles 10 ⟩;
   ⟨ Check data consistency 11 ⟩; ⟨ Build tree spanning the vertices 12 ⟩;
   ⟨ Read vertex angles 14 ⟩;
   ⟨ Read a sufficient number of initial vertex positions and of extrapolated
       vertex positions 16 ⟩;
 }
```

Each of the high-level instructions above expands to a bit of C code, defined and explained in some other section of the source file—wherever it seems to belong from the expository point of view. (Sections are automatically numbered by **WEB** for ease of reference.) Here, for example, is the expansion of ⟨ Check data consistency 11 ⟩:

11. Among the simplest checks that we can carry out to verify whether the data just read make sense is Euler's formula: for a convex polyhedron, $n\_faces + n\_vertices = n\_edges + 2$. Furthermore, since all the faces are assumed to be triangles, we must have $n\_edges = 3 * n\_faces/2$, which, together with Euler's formula, gives $n\_vertices = n\_faces/2 + 2$.

```
⟨ Check data consistency 11 ⟩ ≡
 if (n_edges ≠ 3 * n_faces/2) {
   Fprintf(stderr,
       "Incompatible number of edges (%d) and faces (%d)\n",
       n_edges, n_faces); exit(1);
 }
 if (n_vertices ≠ n_faces/2 + 2) {
   Fprintf(stderr,
       "Polyhedron is not a sphere! (%d vertices for %d faces)\n",
       n_vertices, n_faces); exit(1);
 }
```

This code is used in section 7.

At the end of the program **weave** prints an index containing all occurrences of variables more than one letter long, as well as other words or phrases that the user decides should appear in the index. For example, in the portion of the source

corresponding to section 11 (and elsewhere) I included the line `@^consistency checks@>`; here's how the entry appears in the index:

---

complex: 2.
consistency checks: 9, 11, 14.
cos: 46, 66.
cprintf: 62.
cross_product: 26, 37.
csub: 18, 21, 22, 29, 37, 61, 62, 63, 76.
cur_edge: 13, 22, 23, 52, 61.
cur_face: 20, 63.
cur_vertex: 10, 41, 42, 46.

Other noteworthy points in this index sample are underlined references, which are implicitly generated when a variable declaration is found in the source file, as in the statement

```
double cross_product();
```

and the boldface entry **complex**, indicating a user-defined variable type. When my version of C **weave** finds code such as

```
typedef struct { double real; double imag; } complex;
```

in the source, it automatically starts treating the identifier **complex** as a type, formatting it in boldface and allowing it in the declaration of new variables.

Another area in which the facilities of C allowed me to innovate was the handling of the intermediate C file. In principle this file is for machine consumption only, so it doesn't have to be nicely formatted; but what happens when the compiler, say, detects an error in your code? The error is a lot easier to find if the compiler informs you of its location in your **WEB** source file, not in the intermediate C file. So when **tangle** is writing the C file it preserves the line breaks from the source and inserts **#line** preprocessor commands wherever necessary to ensure that the compiler knows at each point what line of the source a given line of the C file comes from. This helps with most debuggers, as well (not all debuggers know about **#line**, although they should).

Evaluation

The gain in clarity obtained by using **WEB** over C should now be obvious. Are there any drawbacks? After all, it costs something to write documentation, run a preprocessor (**tangle**), and print a document (**weave**+TEX). The cost of running **tangle** is negligible in my system (about one quarter of the compilation time

of the resulting program). Printing the document takes relatively long (several minutes for the thirty-page program discussed above), but I only do it when I want a *global* view of the program; otherwise I just consult the source file. Finally, I agree with Knuth's experience [1984, section L] that the total time for writing and debugging a program ends up being less, in spite of the added documentation.

C **WEB** does, however, have some shortcomings. The main of them is that it is not as flexible as one might hope: if you don't like something about the formatting of C code, for example, you may have to create a private version of C **WEB** with your own formatting rules (this is not very hard, though). I intend to make the rules user-definable in a future release. C **WEB** should also be more versatile in handling multiple source files, so useful in C programming; although I introduced a **WEB** command to perform file inclusion at the **WEB** level (similar to the `#include` command of C), juggling many source files is still somewhat clumsy.

Distribution

My version of C **WEB** is written in C and should run under any operating system with only minor changes. (The port from UNIX to VMS, done by Bjørn Larsen from the University of Oslo, required four changes.) It comes with source, manual and examples in a directory, which you can get via anonymous FTP from `princeton.edu` (change to directory `pub/cweb`). Alternatively, send a mail message to `levy@princeton.edu` or, if you can't establish a mail link, send a written request, specifying whether you want a half-inch tape, quarter-inch tape, PC or Mac floppy. (For tapes or floppies we charge a nominal handling fee.)

Comments and suggestions about the program are welcome. If you port it to other operating systems, or make changes that are likely to be interesting to other people, please share them with me; they may become part of later releases.

Bibliography

Klaus Guntermann and Joachim Schrod, **WEB** adapted to C. *TUGboat*, **7** (1986), 134–137.

Donald E. Knuth, *The* **WEB** *System of Structured Documentation.* Stanford University Computer Science Report CS980 (1983).

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

Silvio Levy, **WEB** adapted to C: another approach. *TUGboat*, **8** (1987), 12–13.

H. Thimbleby, *Cweb.* Preprint, University of York (1983).

Porting TEX to the IBM RT

RICHARD SIMPSON

IBM Corporation
Austin, Texas

ABSTRACT

Richard discussed his successful port of TEX to the IBM RT workstation. He now has a working INITEX, VIRTEX, LATEX, *AMS*-TEX, and BIBTEX, also preview and print capability.

Richard was not able to submit his contribution to the proceedings before the publication date. It will appear in a later TUGBOAT.

— ED

Text Formatting and the Maryland Lawyer

ALLEN R. DYER, ESQUIRE

2922 Wyman Parkway
Baltimore, MD 21211
(301) 243-7283

ABSTRACT

The Maryland State Bar Association Economics of Law Practice Section is currently involved in a standardization effort involving text formatting of legal documents in the Maryland legal community. The Economics of Law Practice Section has endorsed the TEX environment and is actively encouraging those law firms interested in text formatting to voluntarily standardize on TEX.

Introduction

In April of 1987 the Maryland State Bar Association Economics of Law Practice Section (MSBA Economics Section) voted to endorse TEX as the standard for text formatting within the Maryland legal community. This presentation provides some background to that endorsement, describes some of the goals of the MSBA standardization effort, and seeks to enlist the support of the TEX user community.

The recent introduction of the TEX implementations by PCTEX and Addison Wesley on MS-DOS machines and trade magazine publicity about "desktop publishing" made the Maryland legal community aware of the availability of high quality text formatting. Once the MSBA became aware of the opportunity and capabilities offered, there was little delay before action was taken.

The following chronology gives some idea of the period of time which elapsed from the introduction of TEX usage in the MSBA Planning Subcommittee to the endorsement of TEX and the implementation of a plan to promote widespread

use of the TEX environment.

Nov 86 — First use of TEX in MSBA Planning Subcommittee on Automation

Apr 87 — Endorsement of TEX as standard for Maryland legal community.

May 87 — Submission of Interim Report of Planning Committee to the Board of Governors using TEX.

Jun 87 — Article explaining TEX endorsement appears in The MSBA Bar Journal.

Aug 87 — Creation of Economics Section Text Formatting Subcommittee.

Aug 87 — Tentative plans for a special TEX reference issue of the MSBA Bar Journal in early 1988.

Doubtless there are few TUG members that could not enumerate a dozen reasons why TEX is a superb choice for high quality text formatting. However, the MSBA Economics Section chose TEX as the standard text formatting environment not because of TEX's admittedly superior capabilities but, rather, because of TEX's public domain roots. The MSBA Economics Section Council felt vendor independence was absolutely necessary for any software that was to be strongly recommended for extensive use in the legal process.

Goals

The MSBA text formatting standardization effort seeks improvement in the legal process by the following specific goals:

Better legal documents;

Establishment of the legal community as a worthwhile market for software developers;

Increased communication between the academic community and the legal community;

Experimentation with voluntary MSBA software standards; and,

Stimulation of similar efforts in the bar associations of other states.

Better Legal Documents

The American legal system is heavily dependent upon written documents. Accordingly, a major portion of the man-hours expended in the practice of law involves the planning, creation, and analysis of documents. Therefore, any improvement of the ability of the legal community to deal with documents would have a wide ranging effect on the cost and quality of our legal system.

While the use of TEX obviously produces a better looking legal document there may also be corresponding improvement in the content of the legal document.

Current word processing practice in the law office is heavily dependent on the particular features of the word processing software used by the individual firm. And in most small to medium sized law offices the legal secretary is responsible for selecting a previously prepared document, or set of document clauses, and manually customizing a copy of the previous document by multiple finds and replaces throughout the entire document. The use of TEX would allow a more powerful use of macros and definitions at the beginning of a document to restrict all necessary changes to one location in the document.

An even more positive advance would be the development of automatic document production systems that could use TEX as a backend. For example, an attorney could be presented with series of questions which called for the use of the attorney's legal judgment and then, based upon the attorney's input, an expert system could construct a legal document with appropriate TEX codes for final review by the attorney and submission to the TEX processor.

The Legal Community as a Software Market

There are currently 622,000 attorneys in the United States and 16,000 of these attorneys practice in Maryland. This amounts to a ratio of 1 attorney for every 375 laymen. Therefore, a programmer of a legal application program is initially limiting his/her potential customer base to about one fourth of one percent of the general population. Obviously, any software house developing software specifically for legal applications must be able to make a significant penetration of the legal market or else be forced to price the software out of the reach of the smaller firms.

Furthermore, if the legal market is divided into incompatible groupings (different hardware, operating systems, character codes, etc.) the potential customer base rapidly decreases even more. The MSBA Economics Section hopes to prevent a splintering of the already small legal market by encouraging a text

formatting standard. Accordingly, the cost of developing legal applications could be spread over the greatest number of lawyers.

Communication between Academics and Lawyers

Unlike many of the professions, lawyers are relatively isolated from the academic community. In fact, the adversarial nature of law practice tends to divide lawyers into groups of attorneys dependent upon the clienteles of the attorneys. Such isolation slows the dissemination of new approaches to legal problems and slows the introduction of new technology in the law office.

The MSBA TEX Standardization Project is an effort to provide a common language between academics and lawyers relating to the mechanics of document production. The anticipated benefit to the legal community would be more efficient production of legal documents and access to new information and perspectives for dealing with information based problems. The benefits to the academic community would be greater access to the challenging problems of legal process plus an opportunity to make direct contributions to a process that is increasingly intertwined with day to day life.

Experimentation With Voluntary Software Standards

The MSBA has not attempted to encourage software standardization in the past. This hesitancy to attempt standardization stemmed from many reasons including:

Fear of vendor dependence;

Lack of any stable, mature, unchanging software;

Unwillingness to interfere with the marketplace;

Lack of any truly superior software to endorse; and,

Lack of any perceived benefits from standardization.

As a result, Maryland lawyers have been faced with continuing difficulties in transferring machine readable information between law offices, getting vendor support for problems unique to lawyers, finding properly trained personnel, and sharing practice tips involving the use of computers in the practice of law. These problems tend to get even more complicated since most attorneys have little, if

any, technical training and, therefore, have difficulty describing hardware or software problems.

An especially poignant example is the history of word processing use in Maryland law offices. The last decade of legal word processing in Maryland has seen dozens of "popular" word processing programs; at least a half dozen "standard" operating systems; three major "dedicated stand alone" word processor machines; and a gamut of different physical and electronic media formats for document storage.

The experience of the Maryland Bar with the word processing Tower of Babel has resulted in sufficient frustration with vendor incompatibilities and recurring office personnel retraining costs to justify experimenting with standardization. If endorsement of the TEX environment does reduce the confusion that is seemingly inherent in software selection then consideration can be given to a strong, continuing participation of the MSBA in the shaping of the software environment in the law office.

Promoting Standardization in Other States

The Maryland State Bar Association, through the Economics Section, has done more in the area of text formatting standardization than any other state level bar association. If the project is reasonably successful in Maryland then other state bar associations could be expected to benefit from similar efforts. Accordingly, every effort is being made to organize the MSBA project in a generic fashion that will allow easy transfer of results and materials to other states. The MSBA would benefit not only in prestige but also in an increase in the total legal software market size for TEX related products.

Examples of TEX Usage by Maryland Lawyers

To date, the most visible user of TEX has been the Maryland Office of the Public Defender(OPD). The OPD Death Penalty Section has used TEX and the LaTEX macro package to publish a newsletter to update defense attorneys on recent developments in death penalty law and, also, to petition the Governor of Maryland for commutation of the death sentence of Doris Foster. On February 24, 1987, subsequent to receipt of the TEX formatted petition, Governor Hughes commuted Ms. Foster's death sentence to life imprisonment.

Allen R. Dyer, Esquire

How TEX Users Can Help

Before discussing ways TUG members could help with the MSBA TEX effort, I would like to thank two TUG members, Chris Biemesderfer and Richard Furuta, who have helped get the project to the current stage.

Other TUG members that are interested in participating in this standardization project have a variety of options, including:

1. Writing an article for a special Bar Journal issue. The MSBA Economics Section hopes to convince the Maryland Bar Journal to devote a special issue to text formatting. An Appendix at the end of this article sketches the general content that is desired.

2. Advertisements in the special issue. Vendors of TEX related products can provide the MSBA Economics Section with bargaining power by offering to advertise in the proposed special issue of the Bar Journal. Additionally, Maryland attorneys would always have ready access to vendor information whenever they referred back to the special issue.

3. Macro Development. Assuming a significant number of attorneys decide to move into the TEX environment, there will be a demand for macro packages related to the sometimes arcane area of legal document formatting.

4. Service on MSBA Economics Section Subcommittee. I am looking for individuals willing to participate in a correspondence committee on text formatting. The makeup of the subcommittee will be a mixture of technically oriented attorneys and members of the TEX community. You do not have to be a Maryland resident or an attorney to participate. Meetings will be conducted by mail and/or by electronic messaging (BITNet?).

5. Suggestions. Even if you do not have the time for any of the above committments, any suggestions you have would be greatly appreciated.

For more information please contact: Allen Dyer, Chair of the MSBA Economics of Law Practice Subcommittee on Text Formatting Standardization.

Typesetting by Stephen Bencze of TEXSource, Houston

Appendix

Special Issue - Economics of Law Practice Section

POSSIBLE CONTENTS

1. What is Text Formatting?
   - Standards - Why
   - History of Word Processing in MD Legal Community
   - History of TEX
   - TEX's Legal Status - Trademark, Public Domain, AMS
2. Installing TEX
   - TEX Hardware Requirements
   - Macro Packages for TEX
   - TEX Training
   - Using WS, MS WORD, WordPerfect with TEX
   - TEX Startup at a Medium Size Law Firm
3. Text Formatting Concepts and Legal Practice
   - Levels of Abstraction with TEX
     a. Direct Document Production
     b. Variable Names
     c. Program Driven
   - SGML and TEX
   - Normalization of Legal Language
   - Symbolic Logic as a Legal Reasoning Tool
4. Using TEX in Legal Practice
   - Use of TEX at the OPD
   - Handling Correspondence with TEX and/or LaTEX
   - Using TEX for a Simple Will
   - Using TEX for Pleadings
5. Reference Material
   - Glossary - Computers/Typesetting
   - Vendors List
   - Colophon Article
   - TEX User's Group

Of Metafont and PostScript

Leslie Carr

Department of Electronics and Computer Science
The University
SOUTHAMPTON S09 5NH
U.K.
E-MAIL lac@uk.ac.soton.cm

ABSTRACT

METAFONT, a design language for alphabets, has been used to provide fonts for the TEX family of typesetting systems. POSTSCRIPT, a language for representing graphical objects and text, is provided as an interface to the marking engines of many top-quality printing devices. This paper describes an attempt to coerce METAFONT's font designs into POSTSCRIPT's representation of graphical objects.

All the work was done in a UNIX environment, so the tools used are those indigenous to UNIX, *e.g. sed, yacc* and *lex*. Apologies to those who do not live in a UNIX world.

1. By Way of Introduction

TEX arrived at our site a year ago. A certain amount of work had to be done to make it function with our compiler/machine combination; this became a large amount of work to make TEX use the 13 POSTSCRIPT fonts provided in our first LaserWriter Plus. That work made us curious about the relationship between POSTSCRIPT's fonts and those used by TEX—how were the character representations similar? Several months were spent in trying to coerce METAFONT (TEX's font-creating program) to produce POSTSCRIPT-like font descriptions: this paper documents the results of that work.

1.1 METAFONT for the uninitiated

Knuth descibes METAFONT as

> ... a system for the design of alphabets suited to raster-based devices that print or display text.[1]

METAFONT programs are precise mathematical descriptions of the shapes of letters in various alphabets. These descriptions are *parameterised* so that attributes such as **boldness** or *slantedness* or size can be varied by adjusting the values of the parameters. It is this facility that gives METAFONT its METAness.

1.2 PostScript for the uninitiated

The purpose of PostScript is

> ... to describe the appearance of text, graphical shapes and sampled images on printed pages.[2]

PostScript is used on many printing machines and display devices and is set to become "an industry standard" page description language (PDL). In the PostScript graphics model, text characters are treated as ordinary graphical objects, and are described in the same manner.

1.3 Why bother mixing the two?

PostScript is a PDL whose main concern is with the rendering of text and graphics. The fonts available to it are, for the most part, those licensed to Adobe Systems by the Allied Corporation and the International Typeface Corporation. Using METAFONT as a type design system allows technical/foreign typefaces to be used by any software package on any PostScript device.

Although METAFONT programs describe characters in terms of smooth curves, the characters themselves are output in terms of an array of pixels that must be painted on the physical device. At the moment, this "pixel array" is sent to the PostScript interpreter as data for the **imagemask** command. If the character is magnified, then its digital nature is revealed. However, if original smooth curves that define the character's outline are sent to the PostScript interpreter, then PostScript itself will work out the pattern of pixels to be painted, and any amount of scaling can be done.

---

[1] The METAFONTbook

[2] PostScript *Language Reference Manual*, Addison Wesley

1.4 Scaleable Fonts

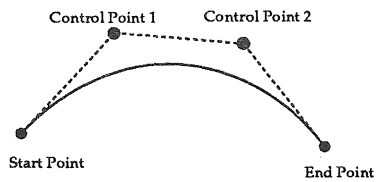
Scaleable fonts have a great advantage — you only need one font description file for all the sizes of that font. At Southampton, we have 23 files devoted to the descriptions of half a dozen sizes and several different magnifications of the *cmr* font.

Traditionally, characters which are to be printed at different sizes are slightly different in their shapes (hence 23 descriptions of the one font!), although Brian Reid (*Unix review*, July 1987, page 55) states that this practise is dying out:

> Essentially all current manufacturers of high-quality phototypesetting equipment use optical scaling on their fonts. Off the top of my head, I can't think of any typesetter manufacturer in the United States that offers more than two versions of a font in a given size and style. Most, in fact, use just one. Although it's true that some little, old, gray-haired men with tweezers—if left to their own devices—would set half a dozen different styles in hot type depending on the magnitude and image they were trying to create, the fact is that technology has simply swept away that sort of thing. The people in the typesetting industry right now, independently of the laser printer market, have stopped using typeface variations across different sizes.

1.5 Curves

Fortuitously, both METAFONT and POSTSCRIPT use Bézier cubics[3] as their method of curve specification. A Bézier curve is completely specified by 4 points—the start point, the end point and two control points. The curve will leave the start point heading for control point 1 and enter the end point heading from control point 2.



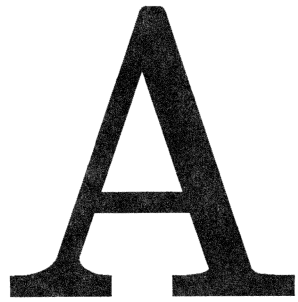
A Bézier curve

METAFONT will automatically calculate the two control points so as to give the "nicest" curve, but POSTSCRIPT requires that all four points be *explicitly* given.

---

[3] See J. D. Foley and A. Van Dam *Fundamentals of Interactive Computer Graphics*, Addison Wesley, for a full treatment of Bézier cubics

2. Turning METAFONT output into PostScript



cmr10 capital A in PostScript

What follows is a summary of the steps that have to be taken to produce a PostScript META font. (The process is completely automated with the command *log2ps*, but this is a breakdown of the steps it follows.) The aim has been to create a PostScript font description from a METAFONT run that is *as normal as possible i.e.* no modifications to the METAFONT program itself or to the files that make up the *plain* and *cmbase* macro packages upon which the cmr fonts are built. The sources of the programs used at each stage are to be found later in the report.

1. The basic idea is to make METAFONT to output a description of all the paths that it traces out in drawing a character. This is done by setting *tracingchoices* to 1. For example, here is a METAFONT file (doit.mf) which will do the job.

```
mode=hires;
tracingchoices:=1;
input cmr10
```

   Typing `mf doit` will produce a generic font file `doit.3000gf` (which can be thrown away!) and a log file (doit.log) that contains all the paths that METAFONT has generated (*hires* is a mode like *luxo* only better, with 3000 pixels per inch).
2. METAFONT prints out the paths before *and* after it has chosen their control points—we only want the final choices. Use *sed*[4] to delete all the blocks headed **paths before choices** to keep those headed **paths after choices**.

```
sed -e '/before choices/,/^$/d' doit.log > doit.after
```

[4] sed is a stream editor which deals with regular expressions

3. METAFONT chops lines up to stop them exceeding 72 characters in length. Unfortunately, it chops numbers across a line! Run *firstaid* on doit.after, saving the results in `doit.aided`

```
firstaid < doit.after > doit.aided
```

   firstaid is a C program generated by *lex*[5] which simply looks for a line that ends with a digit followed by a line that starts with a digit and joins the two together.
4. METAFONT prints out a path description *each time* a path or subpath is evaluated. As the calculation of a path may involve the calculation of many subpaths, it is necessary to delete all those paths which are later incorporated into *super*paths. Run *nocycles* on the result to get rid of these superfluous path descriptions. The result is called `doit.cycles`.

```
nocycles < doit.aided > doit.cycles
```

   Nocycles works very crudely—it simply deletes any path which is non-cyclic on the basis that an area must be cyclic to be filled. This brings problems, as we will see below.
5. Using *paths*, a C program produced by *lex* and *yacc*[6], translate each character's path data into a group of *curveto* and *fill* commands suitable for PostScript. *paths* assumes that each character's description is made up of a number of filled regions, and each region is bounded by a cyclic set of Bézier curves.

```
paths < doit.cycles > doit.paths
```

6. METAFONT simply prints out the paths that it follows without any indication of what it is doing with those paths. Most of the time they are acting as the boundaries of a region to be filled—some of the time they act as boundaries of a region to be *erased.* There are two situations to be watched out for here, one is the use of plain's *erase* macro and the other is when *penstroke* is being used on a cyclic path.
   For this reason, two modifications were made to plain.mf

   a. the string **ERASED** is output when the *erase* macro is called
   b. the string **FILLEDCYCLE** is output when the *penstroke* macro is called on a cyclic path (*e.g.* in the characters O,o and Q)

   Paths marked with **ERASED** are dealt with by the *paths* command (previous) by using *unfill* instead of *fill*, but paths marked **FILLEDCYCLE**

[5] A lexical analyser generator
[6] Yet Another Compiler Compiler

are dealt with at this stage.
When the string **FILLEDCYCLE** is encountered, the last two cyclic paths describe the inside and outside of the region to be filled. Make sure that the first path does not have a separate *fill* command attached to it, and put the PostScript *eofill* command at the end of the second path. This will ensure that PostScript finds out which of the regions is on the outside, and which is on the inside.
This function is provided by an *ed*[7] script, *filledcycles*.

```
filledcycles < doit.paths > doit.chars
```

7. You are now left with a file of character definitions. All that is necessary to turn it into a downloadable PostScript font is to fill in the auxilliary font information that PostScript requires.

- Fill in the Metrics dictionary from the PL file. It would be possible to alter plain.mf so that the *beginchar* macro would output this automatically.
- Fill in the font's Encoding Vector. Currently each character is named **Char-n**, where n is its ASCII code. In this case the encoding vector is trivial.
- Fill in a suitable value for the FontMatrix array. This can be obtained from the *currenttransform* variable during the execution of the *modesetup* macro. This will tell PostScript the size of the co-ordinate system used to define the character, and also any slanting applied to oblique or italic fonts. Please note that any transforms are applied to METAFONT's paths *after* they have been calculated, so a slanted font (*e.g.* cmsl10) would come out straight without transferring *currenttransform* into FontMatrix. This method will not work with some fonts (such as *maths italic*) which do not have all of their characters slanted.

  This step has not yet been completely automated as NeWS has no font operators to speak of.

3. The Problems

The current problems fall into two divisions—practical and theoretical.

[7] similar to *sed*, but works on files instead of streams

3.1 Practical Problems

The major problem encountered is that of *resolution*. METAFONT is designed to do *intelligent* digitisation for low resolution devices, whereas POSTSCRIPT is *not*. All METAFONT's curves are massaged to give the best fit for the underlying device pixels, knowing that the pixel-pattern will never change size or orientation with respect to the device raster. The POSTSCRIPT outline characters are *meant* to be scaled and rotated—and unfortunately most POSTSCRIPT devices fall into the low-resolution category. The result is that trying to use a POSTSCRIPT version of the cmr10 characters on an Apple LaserWriter leaves them blotchy and uneven—some strokes are too thick, some too thin and curves have ugly spots and pimples on them. This problem should diminish with the use of high-resolution devices such as the Linotronic 300 Imagesetter.

Another problem encountered is the sheer size of the POSTSCRIPT character definitions generated. Each character has dozens of subpaths (a cmr M has over 60), each of which requires 6 parameters (two co-ordinates for each of two control points and an end point), leading to over 400 POSTSCRIPT tokens to describe the character. The Apple LaserWriter has a limit of 500 items on its operand stack; this is exceeded by many character descriptions so the procedures for each character are split into subprocedures each 300 tokens long by the *paths* command.

3.2 Theoretical Problems

The major barrier to the use of this method is METAFONT's use of pens. METAFONT has two types of pens—fixed pens and flexible pens. The fixed pen is used with a *pickup* command and has a constant shape throughout the path it draws; the flexible pen varies its width and angle of inclination throughout the path it draws by use of the *penpos* command. The fixed pens are part of 'bare' METAFONT, the flexible pens are a fiction of the plain macro package, implemented as a cyclic path which outlines the stroke of the flexible pen.

All paths are stroked out with one of METAFONT's *fixed* pens; this fixed pen overlaps the path that is stroked out, effectively adding half of its breadth to the region that is to be filled. Altering the width of this pen will alter the shapes of the corners—a large pen will give round corners, smaller pens will give sharp corners. In the cmr10 font, the *crisp* pen has diameter zero, so serifs have square corners. In the cmtt10 font, *crisp* is set to a larger value and the serifs end in semicircles. Because the shape of the current pen (not necessarily circular!) can NOT be taken into account in POSTSCRIPT, these differences in the character shapes will not be seen.

This is a **fundamental** problem: given a path p and a pen q (whose shape is also an arbitary path), METAFONT effectively envelopes p with respect to the shape of q; POSTSCRIPT can do nothing other than stroke it to produce a line of constant width. This incompatibility comes to light when the width of the pen is significant to the shape of the character.

Flexible pens are used for most of the cmr characters, but some maths symbols are defined purely in terms of the strokes of fixed-width pens. At the moment, the conversion process will not deal with them properly as they are not the result of filling a cyclic path.

4. The way forward?

The aim of this work has been to alter METAFONT and its macro packages as little as possible. Perhaps delving into the innards of the METAFONT program itself and liberating the exact information that POSTSCRIPT needs is the only way forward. It could also provide true POSTSCRIPT META fonts, where not only the size, but also the boldness and sans-serifness is stated as an argument to POSTSCRIPT's 'makefont' command.

5. The Sources

Here are the sources for the commands in section 2.

5.1 firstaid.l

```
%%
                        char *ch;
[0-9\.]\n[0-9\.)]       {for(ch=yytext; *ch; ch++)if(*ch!='\n')putchar(*ch);}
```

5.2 nocycles

```
nocycles1 | sed -e '/^Path.*[^e]$/d' | tr '!' '\012'
```

5.3 nocycles1.l

```
%%
choices:\n              printf("choices:!");
\n\ \.\.                printf("!..");
```

5.4 paths.l

```
P1                              "Path at line "
P2                              ", after choices:"
%%
This\ is\ METAFONT.*\n\*\*.*    {lineno=3;}
Font\ metrics\ written\ on.*     {/* nothing */}
Output\ written\ on.*            {/* nothing */}
\([a-zA-Z0-9_]+\.mf(\))?        {/* nothing */}
[ \t]                           {/* nothing*/}
\n                              lineno++;
ERASED                          return(ETEXT);
{P1}[0-9]+{P2}                  return(PTEXT);
"controls"                      return(COTEXT);
"and"                           return(ATEXT);
"cycle"                         return(CYTEXT);
[-+]?[0-9]+(\.[0-9]+)?          {sscanf(yytext,"%f",&val); return(NUMBER);}
\(                              return(LPAR);
\)                              return(RPAR);
\[[0-9]+\](\))?                 {sscanf(yytext,"[%d]",&chnum); return(CHARNO);}
\.\.                            return(DOTDOT);
\,                              return(COMMA);
```

5.5 paths.y

```
%token  PTEXT COTEXT ATEXT CYTEXT NUMBER LPAR RPAR DOTDOT COMMA CHARNO ETEXT
%%
font:           many_chardefs
        ;

many_chardefs:  chardef
        |       many_chardefs chardef
        ;

chardef:        {printf("{ {\n"); n=0;}
                many_paths CHARNO
                {if(n % MAXELEMENTS != 0)printf("} exec\n");
                 printf("} /Char-%d exch def\n", chnum);}
        ;

many_paths:     path
        |       many_paths path
        ;

path:           path1 ETEXT {if(cycle){
                                printf("closepath unfill\n");
                                addtostack(3);
                                }
                        }
        |       path1 {if(cycle){
                                printf("closepath fill\n");
                                addtostack(3);
                                }
                        }
        ;

path1:          PTEXT first_coord {fx=x; fy=y; printf(" moveto\n");
                                   addtostack(1); }
                other_coords
        ;

first_coord:    coord
        ;

other_coords: the_rest
```

```
        | other_coords the_rest
        ;

the_rest: DOTDOT COTEXT coord ATEXT coord DOTDOT last_coord
        {printf("curveto\n"); addtostack(1);}
        ;

last_coord: coord
        | CYTEXT        {x=fx; y=fy; cycle=1; printf("%f %f ", x, y);
                         addtostack(2);}
        ;

coord: LPAR NUMBER {x=val;} COMMA NUMBER {y=val;} RPAR  {printf("%f %f ",x,y);
                                                         addtostack(2);}
        ;
%%
float val, x,y,fx,fy;
int chnum,cycle,lineno,n;
char cmdname[20];
#include "lex.yy.c"
#define MAXELEMENTS 300

yyerror(s)
char *s;{
        fprintf(stderr,"%s: %s on line %d\n", cmdname, s, lineno);
        exit(1);
        }

addtostack(d)
int d;{
        int i;
        for(i=1; i<=d; i++)
                if(++n % MAXELEMENTS == 0)printf("} exec\n{ ");
        }

main(argc,argv)
int argc;
char **argv;{
        strcpy(cmdname,argv[0]);
        printf("%%!\n/unfill {1 setgray fill 0 setgray} def\n");
        return(yyparse());
        }
```

5.6 filledcycles

```
#!/bin/sh
TMP=/tmp/fc.$$
cat > $TMP
while fgrep FILLEDCYCLE $TMP >/dev/null
do
ed $TMP >/dev/null 2>/dev/null <<EOF
/FILLEDCYCLE/s///
s/fill/eofill/
?fill?
s/fill/%No filling/
w
EOF
done
cat $TMP
rm $TMP
```

5.7 log2ps

```
#!/bin/sh
sed -e '/before choices/,/^$/d' $1 |
firstaid |
nocycles1 | sed -e '/^Path.*[^e]$/d' | tr '!' '\012' |
paths |
filledcycles
```

5.8 plain.mf

The definition of *erase* has been changed to the following.

```
def erase text t = begingroup interim default_wt_:=_;
 cullit; t withweight _; cullit;
 message "ERASED"; endgroup enddef;
```

The definition of *cyclestroke* has been changed as follows.

```
def cyclestroke_ =
 begingroup interim turningcheck:=0;
 message "FILLEDCYCLE";
 addto pic_ contour path_.l.t_ withweight 1;
 ...
```

Participants, 1987 TEX Users Group Meeting

UNIVERSITY OF WASHINGTON, SEATTLE, 24–26 AUGUST 1987

Notes: 169 participants;
* indicates exhibitor

Robert A. Adams
University of British Columbia
Vancouver, B.C., Canada

Walter Andrews
University of Washington
Seattle, Washington

Bernadette V. Archuleta
Los Alamos National Laboratory
Los Alamos, New Mexico

* **David Babcock**
Personal TEX, Inc.
Mill Valley, California

William W. Babcock
Northern Michigan University
Marquette, Michigan

* **James K. Bailey**
Personal TEX, Inc.
Mill Valley, California

Ronna Bailey
National Center for
Atmospheric Research
Boulder, Colorado

David T. Barfoot
The Open University
Dorset, England

David Barnes
Oregon Software, Inc.
Portland, Oregon

Elizabeth M. Barnhart
TV Guide
Radnor, Pennsylvania

Stephan v. Bechtolsheim
TEX Users Group
Providence, Rhode Island

Nelson Beebe
University of Utah
Salt Lake City, Utah

Barbara N. Beeton
American Mathematical Society
Providence, Rhode Island

Gary S. Benson
Los Alamos National Laboratory
Los Alamos, New Mexico

Eric Berg
Stanford University
Stanford, California

* **Randolph A. Best**
Digital Composition Systems
Phoenix, Arizona

Chris Biemesderfer
Space Telescope Science Institute
Baltimore, Maryland

Cliff Binstock
Great-West Life
Englewood, Colorado

William Black
University of Oxford
Oxford, England

* **Mark Bloore**
FTL systems Inc.
Toronto, Ont., Canada

Jeffrey Boes
Lear-Siegler
Grand Rapids, Michigan

Eileen Boettner
National Center for Atmospheric Research
Boulder, Colorado

Susan Brooks
The Open University
Milton Keynes, England

Virginia A. Brower
Stanford Linear Accelerator Center
Stanford, California

Mimi Burbank
Supercomputer Computations
Research Institute
Florida State University
Tallahassee, Florida

* **Lance Carnes**
Personal TEX, Inc.
Mill Valley, California

Margot Casey
Los Alamos National Laboratory
Los Alamos, New Mexico

Luigi Cerofolini
University of Bologna
Bologna, Italy

S. Bart Childs
Texas A & M University
College Station, Texas

* **Jill Colantuone**
Addison-Wesley Publishing Co.
Reading, Massachusetts

Arvin C. Conrad
Menil Foundation
Houston, Texas

Jeffrey L. Copeland
Interactive Systems Corp.
Santa Monica, California

Mary Coventry
University of Washington
Seattle, Washington

Michael Crampin
The Open University
Milton Keynes, England

John Crawford
Ohio State University
Columbus, Ohio

Jackie Damrau
University of New Mexico
Albuquerque, New Mexico

Michael Doob
University of Manitoba
Winnipeg, Manitoba, Canada

Karl Dusenbury
Lawrence Livermore National Lab
Livermore, California

Allen R. Dyer
Computer Law Laboratory
Baltimore, Maryland

Robert Elliott
EG & G Energy Measurements, Inc.
Las Vegas, Nevada

Maureen V. Eppstein
Stanford University
Stanford, California

Shawn Farrell
McGill University
Montreal, Que., Canada

Michael J. Ferguson
INRS Télécommunications
Verdun, Que., Canada

Barbara Forrest
Los Alamos National Laboratory
Los Alamos, New Mexico

Jim Fox
University of Washington
Seattle, Washington

* **Frank C. Frye**
Computer Composition Corp.
Madison Heights, Michigan

Donna Gardner
University of Washington
Seattle, Washington

Thaddeus A. Gerards
The Open University
Heerlen, Netherlands

Helen M. Gibson
The Wellcome Institute
London, England

Regina M. Girouard
American Mathematical Society
Providence, Rhode Island

Kari E. Gluski
University of Michigan
Ann Arbor, Michigan

James L. Godwin
Library of Congress
Washington, D.C.

Donald H. Goldhamer
University of Chicago
Chicago, Illinois

* **Gail Goodell**
Addison-Wesley Publishing Co.
Reading, Massachusetts

Yas Gotoh
dit Company, Ltd.
Tokyo, Japan

Raymond E. Goucher
TEX Users Group
Providence, Rhode Island

John S. Gourlay
Ohio State University
Columbus, Ohio

* **Gayla Groom**
Kellerman & Smith
Portland, Oregon

Antonio F. Gualtierotti
IDHEAP
Lausanne, Switzerland

Dean Guenther
Washington State University
Pullman, Washington

Hope Hamilton
National Center for Atmospheric Research
Boulder, Colorado

Sherry P. Haney
Martin Marietta Energy Systems
Oak Ridge, Tennessee

Ann A. Hanson
Talaris Systems, Inc.
San Diego, California

Marvin V. Harlow
Los Alamos National Laboratory
Los Alamos, New Mexico

Jim D. Harmon
Re/Spec Incorporated
Albuquerque, New Mexico

Martin V. Harriman
Intel Corp.
Los Gatos, California

* **Martin Havlicek**
Kellerman & Smith
Portland, Oregon

Jörg S. Heinemann
Siemens AG
München, Germany

Douglas R. Henderson
University of California
Berkeley, California

* **Amy K. Hendrickson**
TEXnology, Inc.
Brookline, Massachusetts

Mildred H. Hoak
Los Alamos National Laboratory
Los Alamos, New Mexico

John D. Hobby
AT & T Bell Laboratories
Murray Hill, New Jersey

Alan Hoenig
City University of New York
New York, N. Y.

Sandy Honken
Ohio State University
Columbus, Ohio

Anita Z. Hoover
University of Delaware
Newark, Delaware

Doris T. Hsia
Stanford University
Stanford, California

* **Janice Hughes**
Addison-Wesley Publishing Co.
Reading, Massachusetts

Nancy M. Hunt
Sandia National Labs
Livermore, California

Rjay R. Ilg
dit Company,Ltd.
Tokyo, Japan

Patrick D. Ion
Mathematical Reviews
Ann Arbor, Michigan

Calvin W. Jackson
Calif. Institute of Technology
Los Angeles, California

* **Peter Jacobsen**
DocuSoft Publishing Technologies, Inc.
Vancouver, B.C., Canada

* **Katherine R. Johnson**
AM Varityper
Kent, Washington

Regina A. Johnson
Los Alamos National Laboratory
Los Alamos, New Mexico

Yvonne Johnson
Los Alamos National Laboratory
Los Alamos, New Mexico

Abdo R. Jooya
The Open University
Heerlen, The Netherlands

Helmut Jürgensen
University of Western Ontario
London, Ont., Canada

* **William Kastor**
Personal TEX, Inc.
Mill Valley, California

* **David Kellerman**
Kellerman & Smith
Portland, Oregon

* **David A. Kennedy**
DocuSoft Publishing Technologies, Inc.
Vancouver, B.C., Canada

* **Robert L. Kister**
K-Talk Communications
Columbus, Ohio

Kazuhiro Kitagawa
Keio University
Yokohama, Japan

Carol Klos
Stratus Computer, Inc.
Marlborough, Massachusetts

William J. Kollar
MRJ Incorporated
Oakton, Virginia

Richard B. Lane
University of Montana
Missoula, Montana

Steen Larsen
UNI-C Aarhus
Aarhus, Denmark

Scott Larson
The Davis Group
Seattle, Washington

Dan C. Latterner
Mathematical Reviews
Ann Arbor, Michigan

J. S. Lee
Northrop Corp.
Palos Verdes Peninsula, Calif.

Silvio Levy
Princeton University
Princeton, New Jersey

Betty Lim
Applicon
Ann Arbor, Michigan

Chin S. Lin
Southwest Research Institute
San Antonio, Texas

Doug Lind
University of Washington
Seattle, Washington

Pierre MacKay
University of Washington
Seattle, Washington

Lucille Maestas
Los Alamos National Laboratory
Los Alamos, New Mexico

Laurie D. Mann
Stratus Computer, Inc.
Marlborough, Massachusetts

Mary W. Marler
Vanderbilt University
Nashville, Tennessee

Carole D. McCartney
TV Guide
Radnor, Pennsylvania

Robert W. McGaffey
Oak Ridge National Lab
Oak Ridge, Tennessee

Claudia H. McNellis
Library of Congress
Washington, D.C.

Marie McPartland-Conn
GTE Laboratories
Waltham, Massachusetts

Robert A. Messer
Albion College
Albion, Michigan

Tom Milliman
University of New Hampshire
Durham, New Hampshire

Patricia A. Monohon
University of Washington
Seattle, Washington

* **Jerry Nakao**
AM Varityper
Kent, Washington

Norman Naugle
Texas A & M University
College Station, Texas

David Ness
TV Guide
Radnor, Pennsylvania

Albert Nijenhuis
Seattle, Washington

Stephanie S. O'Hara
University of Maryland,
Baltimore County
Catonsville, Maryland

Anthony Parks
Varityper
Randolph, New Jersey

David Parmenter
Digital Equipment Corp.
Nashua, New Hampshire

John Peterson
University of Washington
Seattle, Washington

Noel Peterson
Library of Congress
Washington, D.C.

Craig R. Platt
University of Manitoba
Winnipeg, Manitoba, Canada

Mark R. Probert
EG & G Energy Measurements, Inc.
Goleta, California

Sebastian P. Rahtz
Southampton University
Southampton, England

Doug Ravenel
University of Washington
Seattle, Washington

Phyllis J. Renzetti
U.S. Geological Survey
Reston, Virginia

Norman Richert
University of Houston
Houston, Texas

James M. Roberts
Univ. of California, San Diego
La Jolla, California

* **David L. Rodgers**
ArborText, Inc.
Ann Arbor, Michigan

Eugene S. Rodolphe
New York University
New York, N. Y.

Chris A. Rowley
The Open University
London, England

Yoshiteru Sagiya
dit Company, Ltd.
Tokyo, Japan

Nobuo Saito
Keio University
Tokyo, Japan

Yasuki Saito
Nippon Telephone & Telegraph Corp.
Tokyo, Japan

Brian T. Schellenberger
SAS Institute, Inc.
Cary, North Carolina

William Schwarz
State University of New York
Albany, New York

Richard Simpson
IBM Corp.
Austin, Texas

* **Brian Skidmore**
Addison-Wesley Publishing Co.
Reading, Massachusetts

James F. Slagle
TV Guide
Radnor, Pennsylvania

* **Barry Smith**
Kellerman & Smith
Portland, Oregon

Bernd Stümke
Siemens AG
München, Germany

Carol K. Sullivan
U.S. Geological Survey
Menlo Park, California

* **Henry Suwinsky**
Addison-Wesley Publishing Co.
Reading, Massachusetts

Rilla Thedford
Intergraph Corp.
Huntsville, Alabama

Christina A. Thiele
Carleton University
Ottawa, Ont., Canada

* **Dave Thomas**
Digital Composition Systems
Phoenix, Arizona

Margaret M. Thomas
Talaris Systems, Inc.
San Diego, California

David W. Thompson
Lawrence Livermore National Lab
Livermore, California

Peter Thompson
TV Guide
Radnor, Pennsylvania

Ted Toyota
Sony Corp.
Tokyo, Japan

John R. Travis
Software Research Association
Tokyo, Japan

* **Tibor Tscheke**
Univ-druckerei H. Stürtz AG
Würzburg, Germany

Douglas R. Turner
Dept. of Health and Human Services
Atlanta, Georgia

Mary Ann Vigil
Los Alamos National Laboratory
Los Alamos, New Mexico

Kent J. Wada
University of British Columbia
Vancouver, B.C., Canada

Kent A. Wagner
TV Guide
Radnor, Pennsylvania

James W. Walker
University of South Carolina
Columbia, South Carolina

Stacy Waters
University of Washington
Seattle, Washington

Samuel B. Whidden
American Mathematical Society
Providence, Rhode Island

Michael Wiesenberg
Hewlett-Packard
Palo Alto, California

Bruce Wolman
Kaos A/S
Elisenberg, Oslo, Norway

Steve Woods
University of Washington
Seattle, Washington

Jean Wunderlich
The Davis Group
Seattle, Washington

Cheryl R. Wurzbacher
Addison-Wesley Publishing Co.
Reading, Massachusetts

Ralph E. Youngen
American Mathematical Society
Providence, Rhode Island

James P. Zablosky
University of British Columbia
Vancouver, B.C., Canada

* **Lian Zerafa**
FTL systems Inc.
Toronto, Ont., Canada

Note: 169 participants.

The TEXNIQUES SERIES

Publications on TEX and its Environment
Available from the TEX Users Group

1. **VAX Language-Sensitive Editor (LSEDIT) Quick Reference Guide for use with the LATEX Environment and LATEX Style Templates**
2. **Table-Making with INRSTEX** by Michael J. Ferguson
3. **User's Guide to the IdxTEX Program** by R. L. Aurbach
4. **User's Guide to the GloTEX Program** by R. L. Aurbach
5. **Proceedings of the Eighth TEX Users Group Annual Meeting**
   University of Washington, Seattle, Wash., August 24–26, 1987
   Dean Guenther, Editor